Fixed-Point Designer™

Reference

MATLAB®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Fixed-Point Designer™ Reference

Revision History

March 2013	Online only	New for Version 4.0 (R2013a)
September 2013	Online only	Revised for Version 4.1 (R2013b)
March 2014	Online only	Revised for Version 4.2 (R2014a)
October 2014	Online Only	Revised for Version 4.3 (R2014b)
March 2015	Online Only	Revised for Version 5.0 (R2015a)
September 2015	Online Only	Revised for Version 5.1 (R2015b)
October 2015	Online only	Rereleased for Version 5.0.1 (Release 2015aSP1)
March 2016	Online Only	Revised for Version 5.2 (R2016a)
September 2016	Online only	Revised for Version 5.3 (R2016b)
March 2017	Online only	Revised for Version 5.4 (R2017a)
September 2017	Online only	Revised for Version 6.0 (R2017b)
March 2018	Online only	Revised for Version 6.1 (R2018a)
September 2018	Online only	Revised for Version 6.2 (R2018b)
March 2019	Online only	Revised for Version 6.3 (R2019a)
September 2019	Online only	Revised for Version 6.4 (R2019b)
March 2020	Online only	Revised for Version 7.0 (R2020a)
September 2020	Online only	Revised for Version 7.1 (R2020b)
March 2021	Online only	Revised for Version 7.2 (R2021a)
September 2021	Online only	Revised for Version 7.3 (R2021b)
March 2022	Online only	Revised for Version 7.4 (R2022a)
September 2022	Online only	Revised for Version 7.5 (R2022b)
March 2023	Online only	Revised for Version 7.6 (R2023a)

Contents

Apps

Fixed-Point Converter

Convert MATLAB code to fixed point

Description

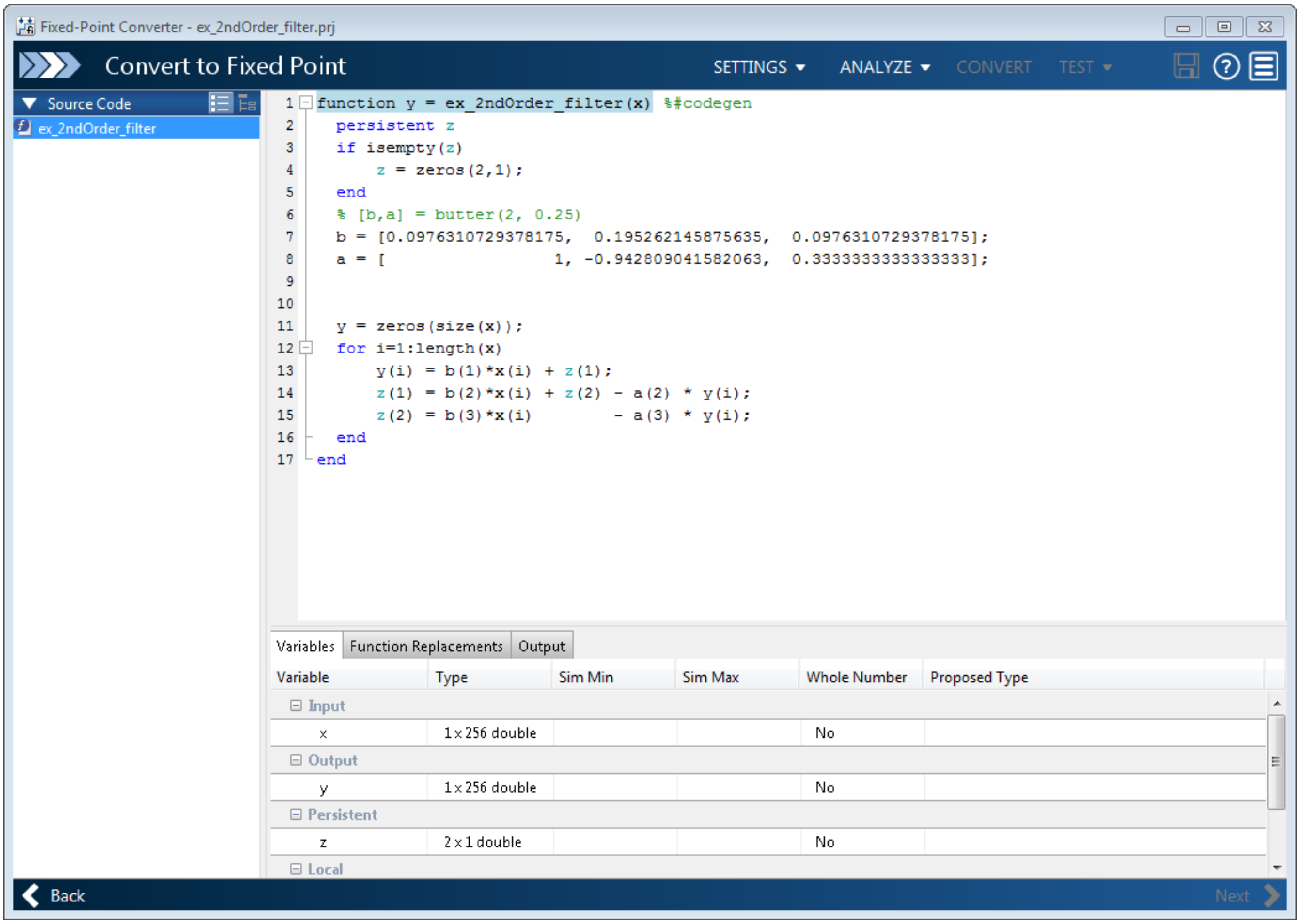
The **Fixed-Point Converter** app converts floating-point MATLAB® code to fixed-point MATLAB code.

Using the app, you can:

- Propose data types based on simulation range data, static range data, or both.
- Propose fraction lengths based on default word lengths or propose word lengths based on default fraction lengths.
- Optimize whole numbers.
- Specify safety margins for simulation min/max data.
- View a histogram of bits used by each variable.
- Specify replacement functions or generate approximate functions for functions in the original MATLAB algorithm that do not support fixed point.
- Test the numerical behavior of the fixed-point code. You can then compare its behavior against the floating-point version of your algorithm using either the Simulation Data Inspector or your own custom plotting functions.

If your end goal is to generate fixed-point C code, use the MATLAB Coder™ app instead. See "Convert MATLAB Code to Fixed-Point C Code" (MATLAB Coder).

If your end goal is to generate HDL code, use the HDL Coder™ workflow advisor instead. See "Floating-Point to Fixed-Point Conversion" (HDL Coder).



Open the Fixed-Point Converter App

- MATLAB Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
- MATLAB command prompt: Enter `fixedPointConverter`.
- To open an existing Fixed-Point Converter app project, either double-click the `.prj` file or open the app and browse to the project file.

  Creating a project or opening an existing project causes any other Fixed-Point Converter or MATLAB Coder projects to close.
- A MATLAB Coder project opens in the MATLAB Coder app. To convert the project to a Fixed-Point Converter app project, in the MATLAB Coder app:

  1. Click  and select **Reopen project as**.
  2. Select `Fixed-Point Converter`.

Examples

- “Propose Data Types Based on Simulation Ranges”
- “Propose Data Types Based on Simulation Ranges”
- “Propose Data Types Based on Derived Ranges”

Programmatic Use

`fixedPointConverter` opens the Fixed-Point Converter app.

`fixedPointConverter -tocode projectname` converts the existing project named `projectname.prj` to the equivalent script of MATLAB commands. It writes the script to the Command Window.

`fixedPointConverter -tocode projectname -script scriptname` converts the existing project named `projectname.prj` to the equivalent script of MATLAB commands. The script is named `scriptname.m`.

- If `scriptname` already exists, `fixedPointConverter` overwrites it.
- The script contains the MATLAB commands to:
  - Create a floating-point to fixed-point conversion configuration object that has the same fixed-point conversion settings as the project.
  - Run the `fiaccel` command to convert the floating-point MATLAB function to a fixed-point MATLAB function.

Before converting the project to a script, you must complete the **Test** step of the fixed-point conversion process.

Version History

Introduced in R2014b

See Also

Functions
`fiaccel`

Topics
“Propose Data Types Based on Simulation Ranges”
“Propose Data Types Based on Simulation Ranges”
“Propose Data Types Based on Derived Ranges”
“Fixed-Point Conversion Workflows”
“Automated Fixed-Point Conversion”
“Generated Fixed-Point Code”
“Automated Fixed-Point Conversion in MATLAB”

Fixed-Point Tool

Convert a floating-point model to a fixed-point model

Description

The **Fixed-Point Tool** enables you to automatically convert a floating-point model to use fixed-point data types, optimize existing data types on a model, and analyze ranges and data types on your model using rich statistics and visualizations.

The **Fixed-Point Tool** provides three workflows depending on your needs:

- **Optimized Fixed-Point Conversion** — Automatically convert your model to use optimized fixed-point data types.
- **Iterative Fixed-Point Conversion** — Automatically propose fixed-point data types and manually select which data types to apply to your model.
- **Range Collection** — Explore the numerical behavior of your model before or after data type conversion.

The table below provides a summary of the differences between these three workflows. These options are explained in more detail below.

Workflow	Changes Model Data Types	Ease of Use	Amount of Control Over Data Types Applied to Model	Requires Knowledge of System Behavior Tolerances	Command-Line Workflow
Optimized Fixed-Point Conversion	Yes	One step	Low	Yes	`fxpopt`
Iterative Fixed-Point Conversion	Yes	Multiple iterations	High	Recommended	`DataTypeWorkflow.Converter`
Range Collection	No	One step	N/A	Recommended	`DataTypeWorkflow.Converter`

Optimized Fixed-Point Conversion Workflow

The **Optimized Fixed-Point Conversion** workflow in the **Fixed-Point Tool** provides a fully-automated means of converting a Simulink® model to fixed point. If you know the desired behavior of your system and can specify acceptable tolerances on this behavior, you can use this workflow to find the optimal data types for your system. You can achieve better results if you additionally specify any known ranges or supply additional simulation inputs.

The tool allows you to specify allowable wordlengths and will also take into account limitations of target hardware you specify. You can also specify a safety margin to increase the bounds of the ranges collected by a specified amount. Optimized data types stay within specified behavioral tolerances and minimize the cost of the design. If more than one feasible solution is found, you can apply and explore different solutions to your model to find one that fits your needs. You can explore the ranges and statistics collected in your baseline model using rich visualization to quickly spot sources of overflow and other numerical issues. You can compare the results of different fixed-point implementations in the Simulation Data Inspector.

After optimizing data types in the **Fixed-Point Tool**, you can export the workflow to a MATLAB script. This allows you to continue data type optimization using `fxpopt` at the command line, which has additional advanced options available for further customizing the optimization process.

This workflow will automatically change the data types on your model at completion of the optimization process. If you complete the preparation step before starting optimization, you can automatically restore your model to its original state.

Iterative Fixed-Point Conversion Workflow

The **Iterative Fixed-Point Conversion** workflow in the **Fixed-Point Tool** is an interactive automatic means of specifying fixed-point data types in a Simulink model. The tool collects ranges for model objects, then proposes fixed-point data types that maximize precision and cover the range. You can then review the data type proposals and apply them selectively to objects in your model.

The tool allows you to propose word lengths or fraction lengths, giving you the option to have a fixed-precision design, and will also take into account limitations of target hardware you specify. You can also specify a safety margin to increase the bounds of the ranges collected by a specified amount. Rich visualizations allow you to explore the ranges of objects in your model and quickly spot sources of overflow and other numerical issues, both before and after converting your model to fixed point. If the proposed data types do not meet your needs, you can continue iterating through this process. You can compare the results of different fixed-point implementations in the Simulation Data Inspector.

This workflow gives you full control over which proposed data types are applied to your model, if any. If you complete the preparation step of conversion, you can automatically restore your model to its original state.

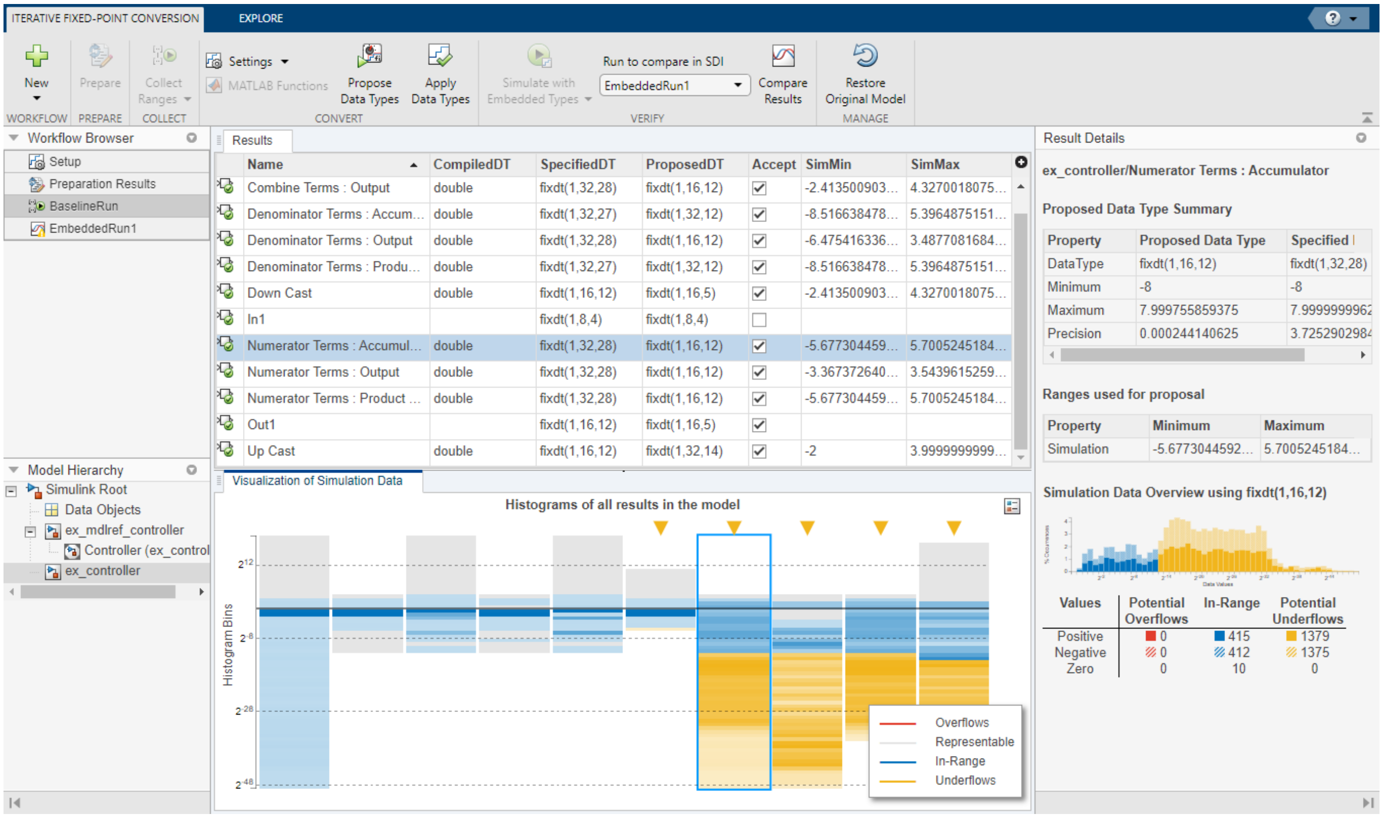
This workflow does not require you to specify the desired behavior of your system, however it is recommended that you specify any known ranges, simulation inputs, and signal tolerances in order to achieve more accurate data type proposals and be able to evaluate whether proposed data types meet the specified requirements of the design.

Range Collection Workflow

The **Range Collection** workflow in the **Fixed-Point Tool** is an analysis and troubleshooting tool, and does not change your model. This workflow provides independent access to the range collection step found in the data type conversion workflows.

You can choose to specify additional simulation inputs and tolerances on logged signals in your model. The tool will individually collect ranges for all simulation inputs specified, and also merge the results for a combined view. If you want to explore the ideal floating-point behavior of your system, you can choose to collect ranges with data type override enabled.

Rich visualizations allow you to explore the ranges of objects in your model and quickly spot sources of overflow, underflow, and other numerical issues, before or after conversion to fixed point. Signals that do not meet the specified tolerances are highlighted in the results. You can compare the results of simulation runs using the Simulation Data Inspector.



Open the Fixed-Point Tool

- Simulink Toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
- MATLAB command prompt: Enter `fxptdlg('system_name')`, where `'system_name'` is the name of the model or system you want to convert, specified as a string.

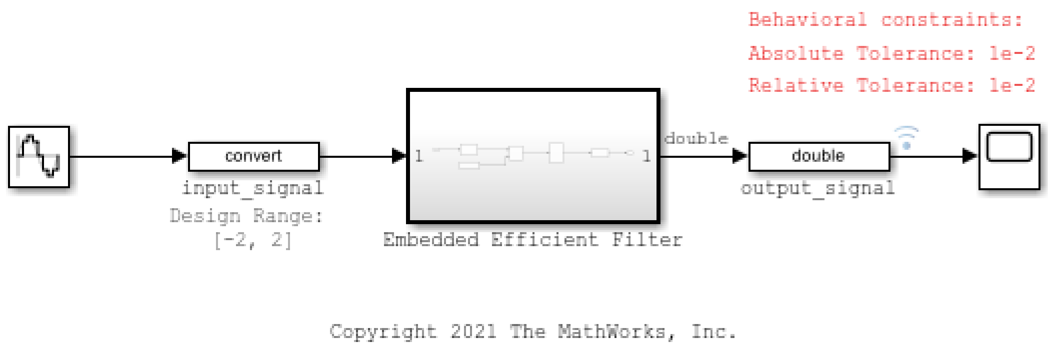
Examples

Optimized Fixed-Point Conversion in the Fixed-Point Tool

This example shows how to use the **Optimized Fixed-Point Conversion** workflow in the **Fixed-Point Tool**. The model used in this example is a simple FIR filter modeled using floating-point data types. In this example, you specify known behavioral constraints for the output of the filter and optimize the fixed-point data types in the `Embedded Efficient Filter` subsystem.

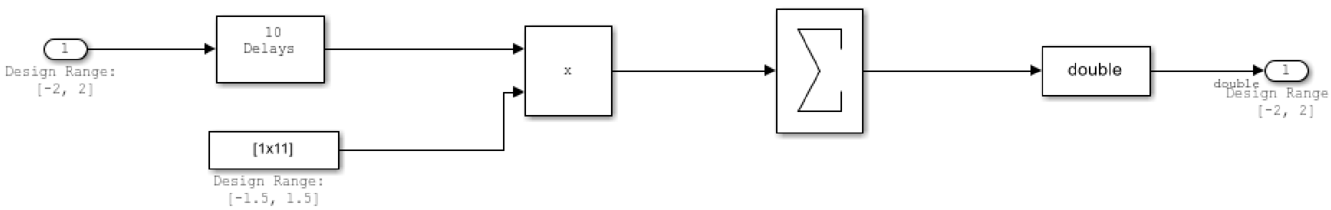
Open the `mSimpleFIR` model.

```
open_system('mSimpleFIR');
```



Inspect the `Embedded Efficient Filter` subsystem.

```
open_system('mSimpleFIR/Embedded Efficient Filter');
```



Known design minimum and maximum values are specified explicitly on blocks in the model, including on the inputs and outputs of the `Embedded Efficient Filter` subsystem.

Open the Fixed-Point Tool. On the Simulink® **Apps** tab, under **Code Generation**, click the app icon.

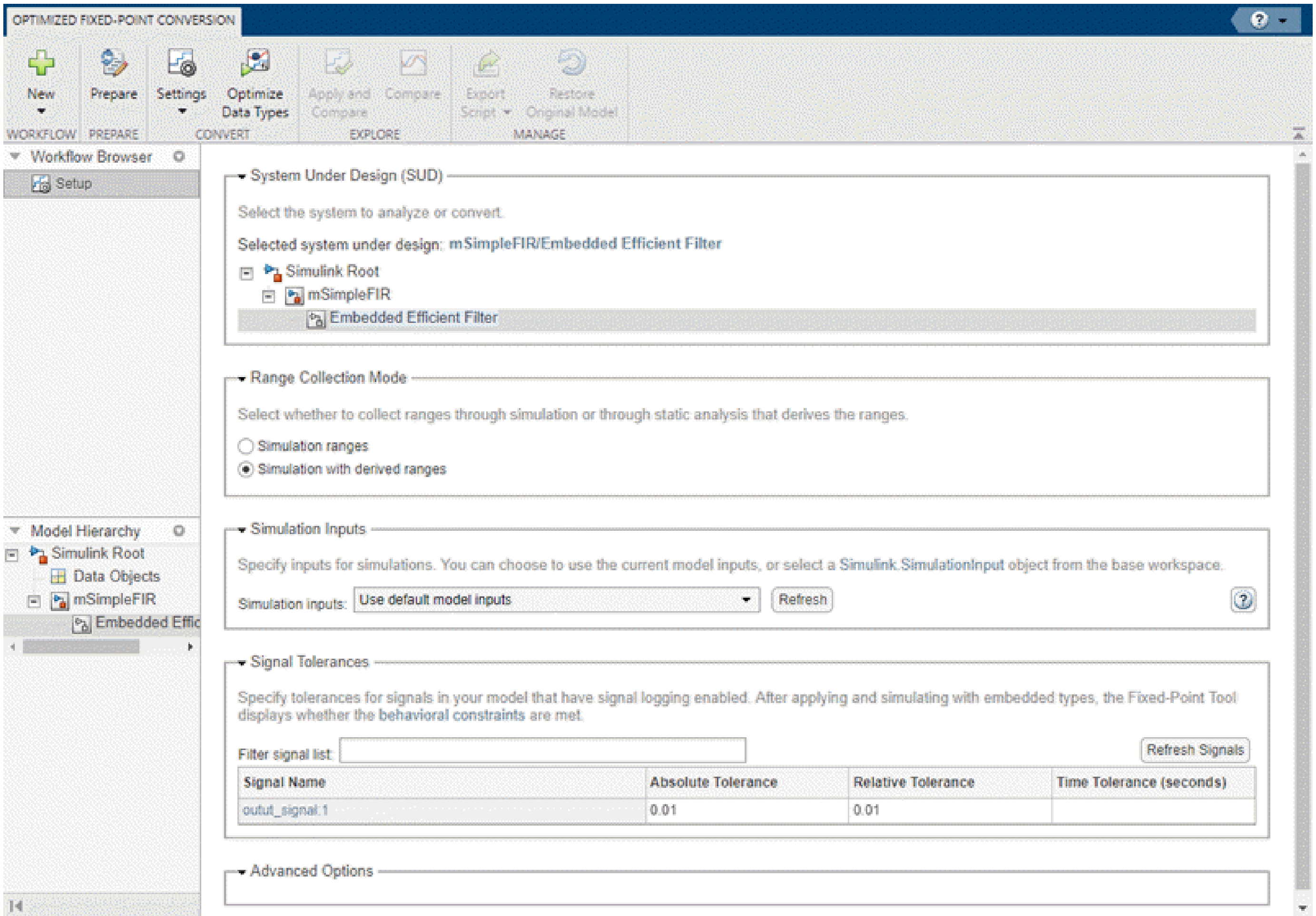
To start the optimized fixed-point conversion workflow, select **Optimized Fixed-Point Conversion**.

Select the subsystem that you want to analyze. Under **System Under Design (SUD)**, select the `Embedded Efficient Filter` subsystem.

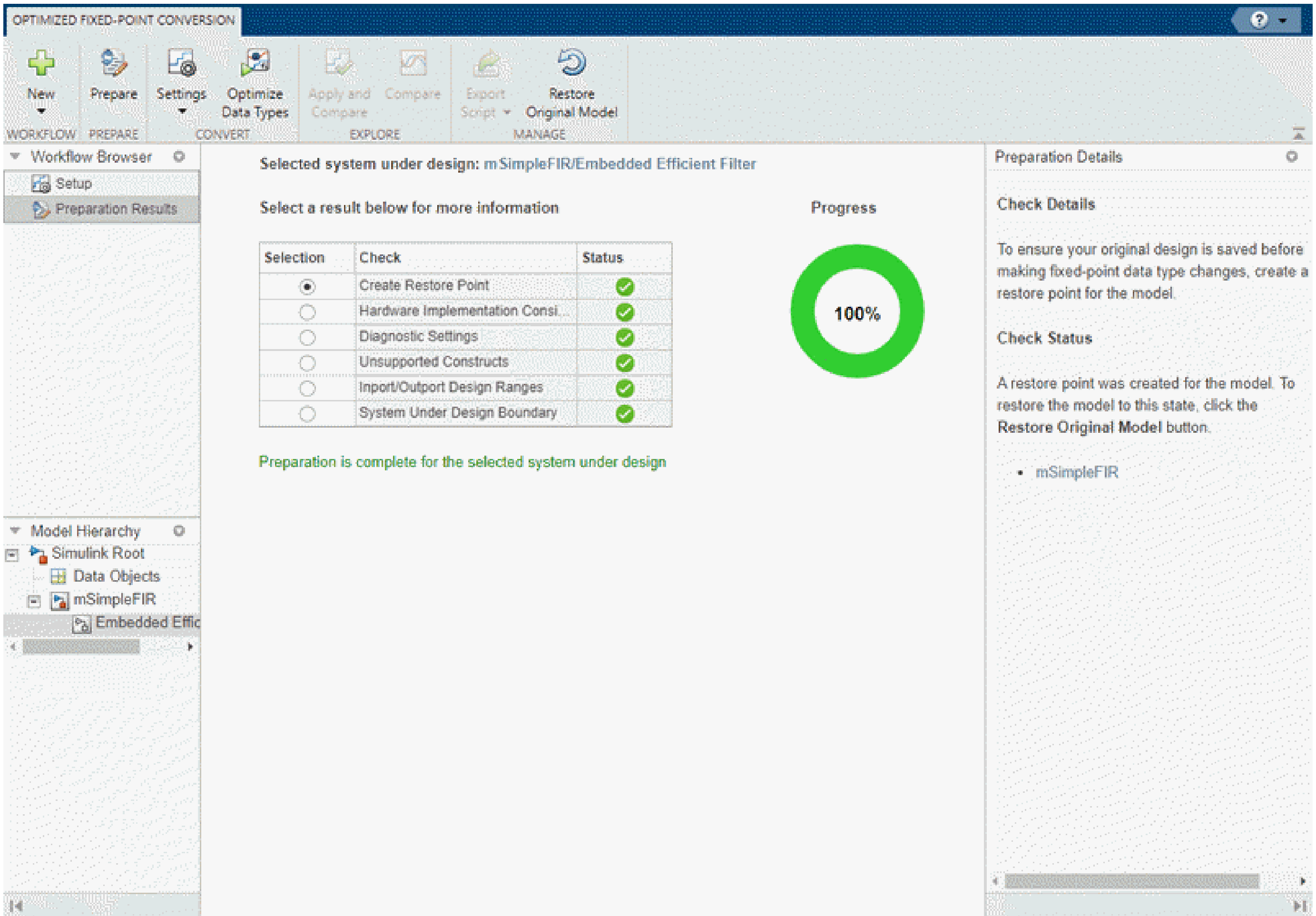
Choose the range collection method to use. Under **Range Collection Mode**, select **Simulation with derived ranges**. During the range analysis step of optimization, the tool will combine ranges from simulation minimum and maximum values, design minimum and maximum values specified explicitly on blocks in the model, and derived minimum and maximum values that are computed through a static analysis that derived ranges for objects in the model.

Specify **Simulation Inputs**. For this example, use the default model inputs for simulation.

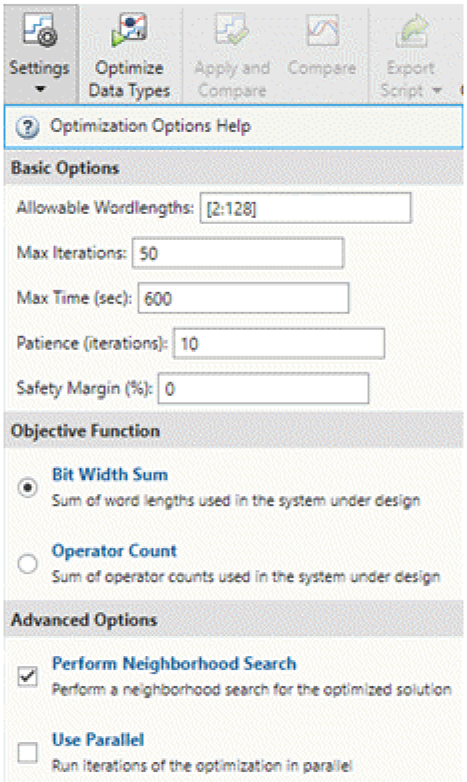
Specify signal tolerances for logged signals. Set the **Absolute Tolerance** and **Relative Tolerance** of the `output_signal:1` to `0.01`.



To prepare the model for fixed-point conversion, click **Prepare**. The Fixed-Point Tool creates a backup version of the model and checks the model for comaptibility with the conversion process. For more about preparation checks, see “Use the Fixed-Point Tool to Prepare a System for Conversion”.



Next, expand the **Settings** button arrow to configure the settings to use for data type optimization. For this example, use the default settings.

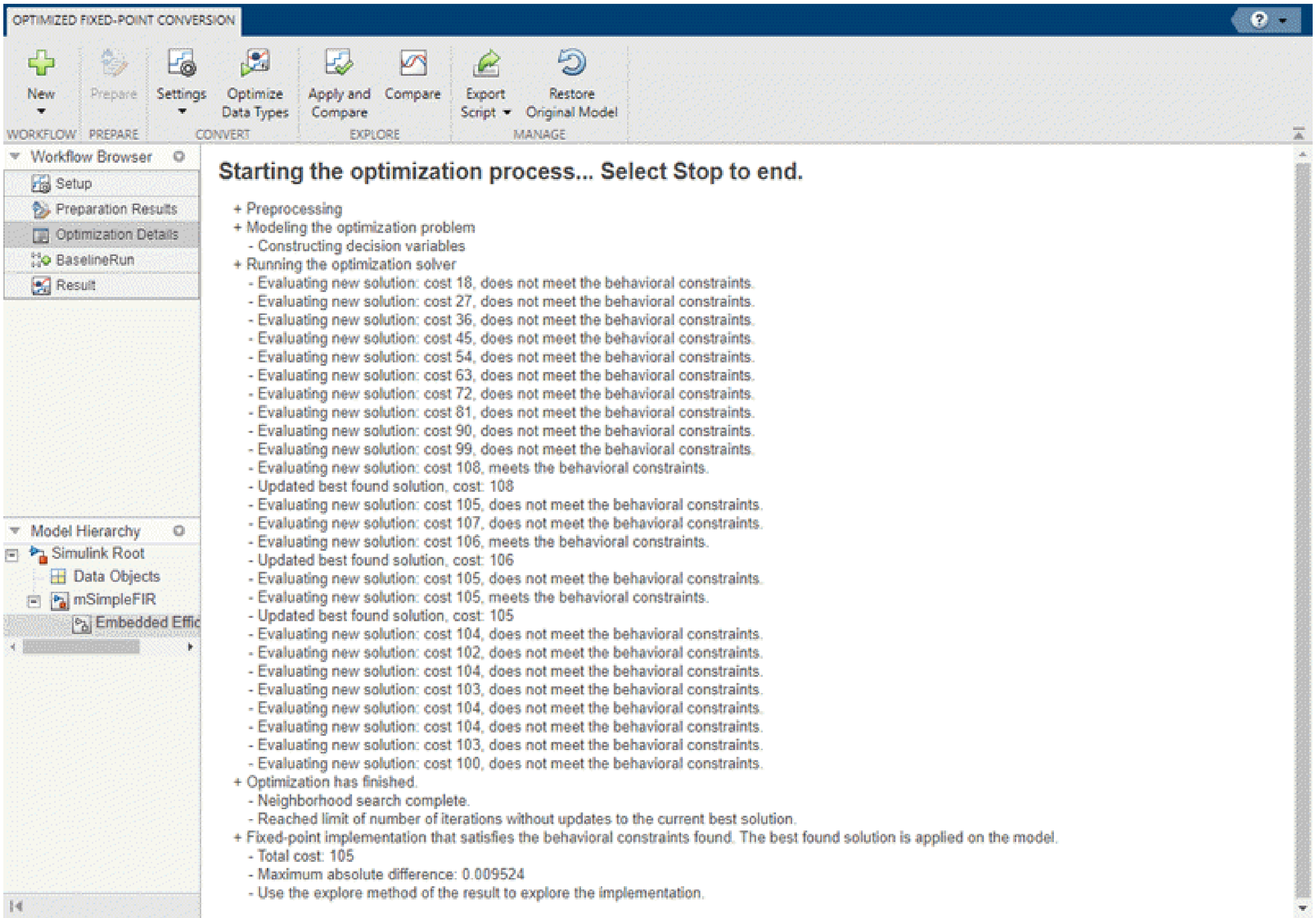


To optimize data types in the model, click **Optimize Data Types**.

During the optimization process, the software analyzes ranges of objects in your system under design. Optimization will take into account all specified behavioral constraints, including design minimum and maximum values and signal tolerances, to apply heterogeneous data types to your system while minimizing the objective function. For this example, the objective function is set to the default `Bit Width Sum`, which instructs the optimization to minimize the sum of word lengths in the system under design.

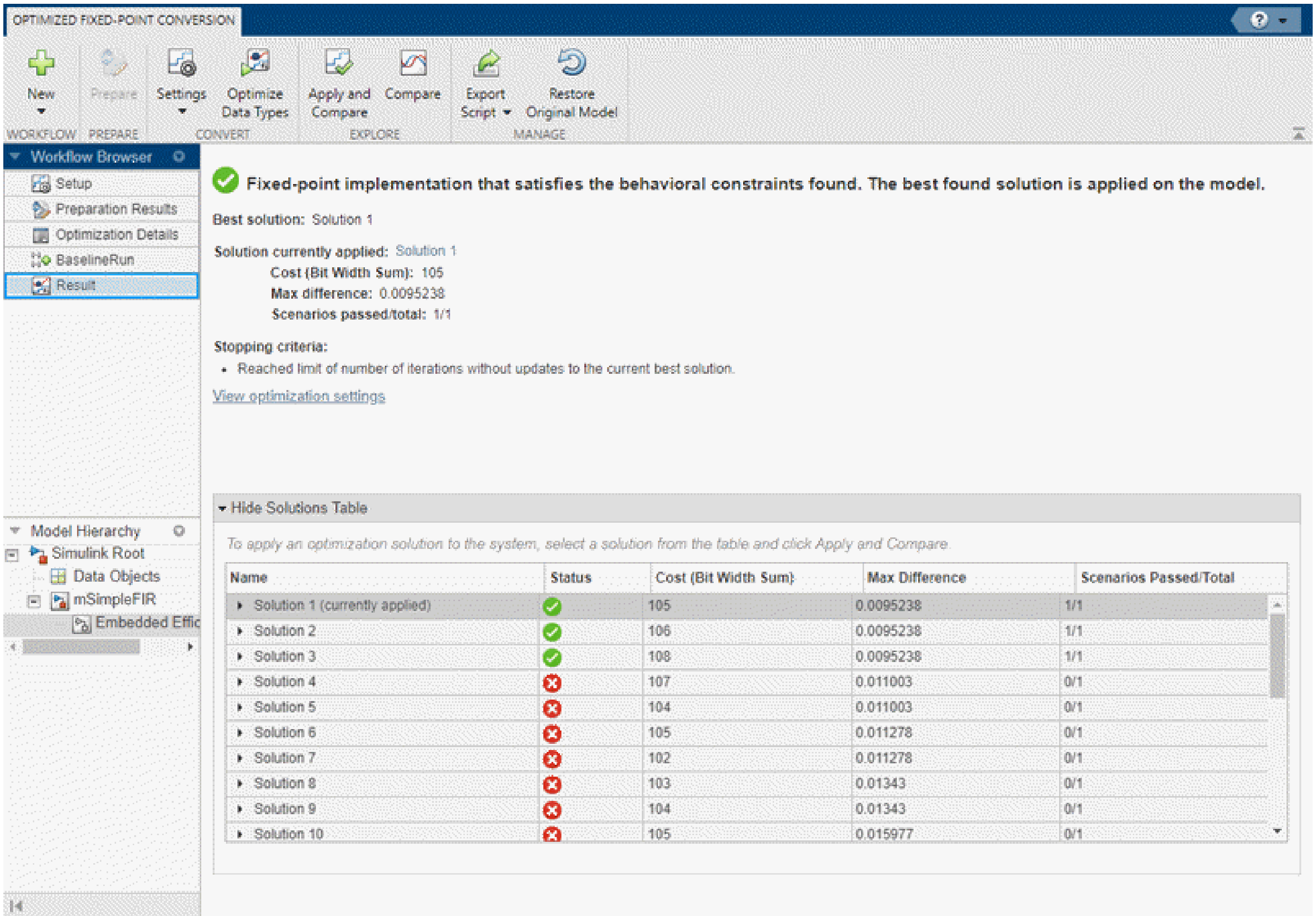
During the optimization process, the software makes changes to several settings and model configuration parameters. These purpose of these changes include suppressing diagnostics, enabling logging with the Simulation Data Inspector, reducing the memory consumed by the result, ensuring validity of the model, accelerating the optimization process, and turning off data type override. For more information, see “Model Configuration Changes Made During Data Type Optimization”. You can restore these diagnostics after the optimization is complete.

Details about the optimization process are printed to the **Optimization Details** pane in the Fixed-Point Tool. You can pause or stop the optimization solver before the optimization search is complete by clicking **Stop**.

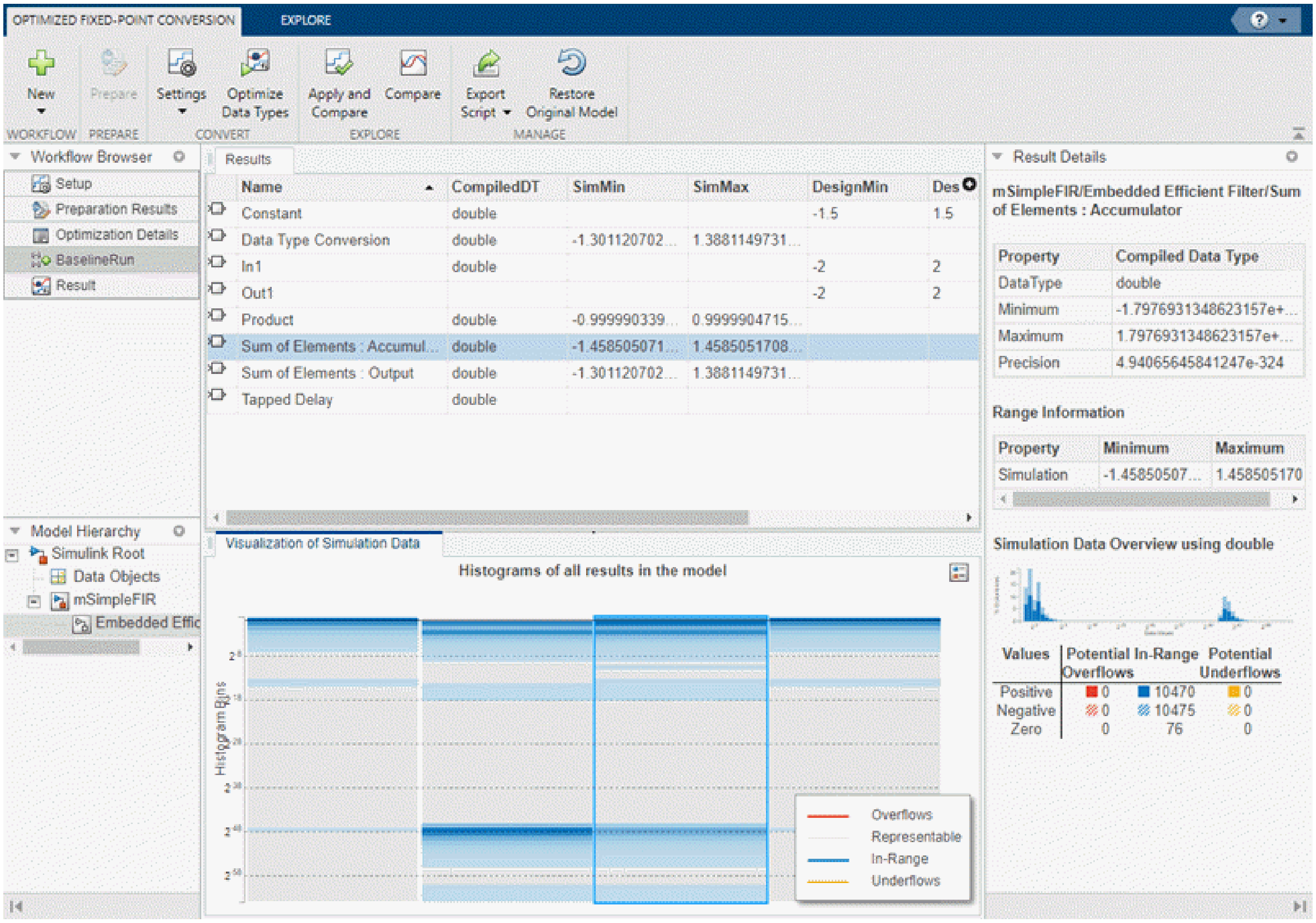


When the optimization is complete, the Fixed-Point Tool displays a table that contains all of the solutions found during the optimization process. **Solution 1** in the table corresponds to the best solution found.

Solutions are ordered in the table based on the **Cost**, which is defined by the objective function specified in the **Settings** menu. Feasible solutions that meet the defined behavioral constraints are marked with a pass status in the solutions table. Solutions that do not meet the behavioral constraints are marked with a fail status. This example uses tolerances on the output of the filter subsystem to define the desired behavior of the system. For more information about defining other types of behavioral constraints, see “Specify Behavioral Constraints”.



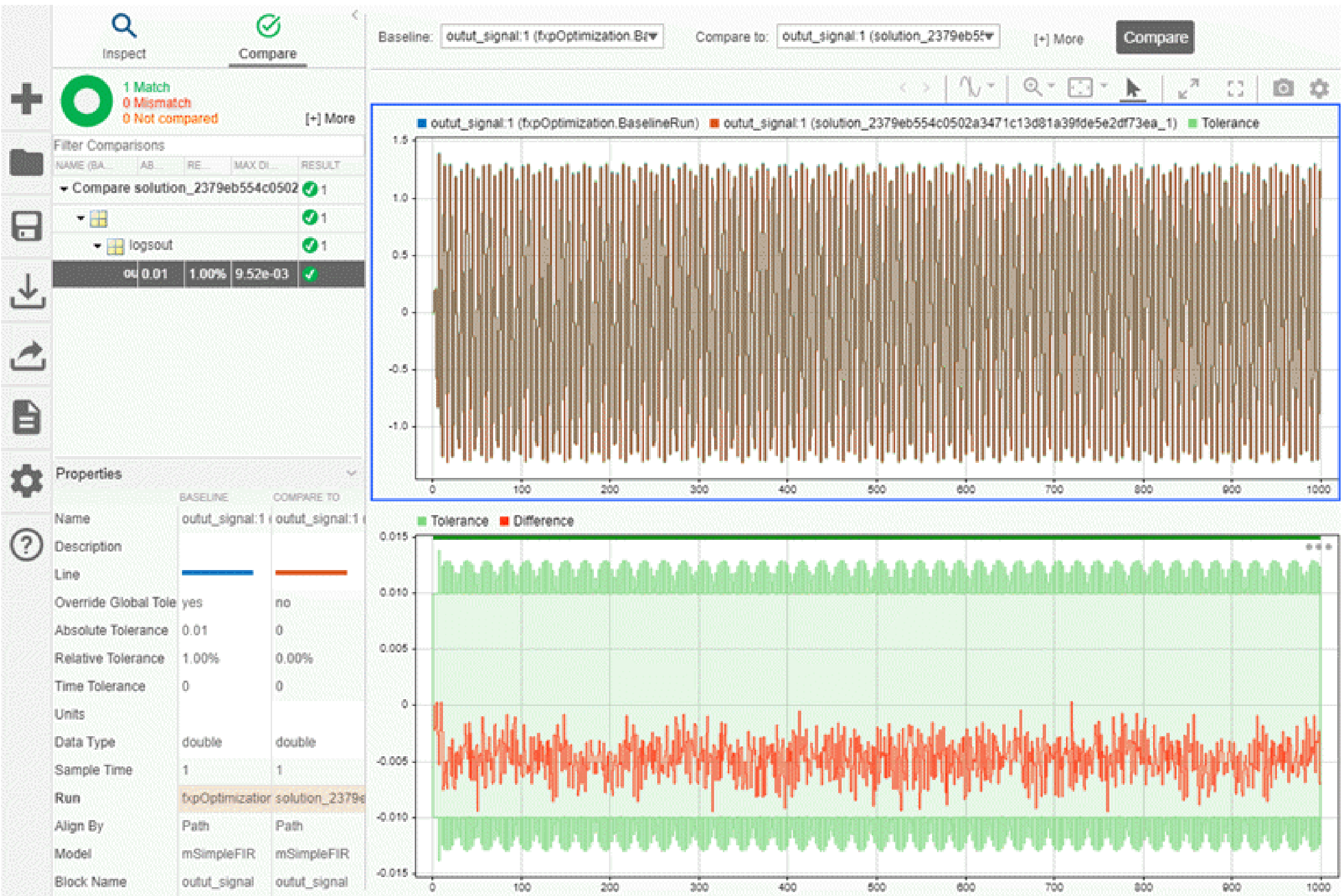
During the optimization process, the tool collects ranges and statistics for objects in your model. To explore these ranges, in the **Workflow Browser** pane, select `BaselineRun`.



The **Results** spreadsheet displays a summary of the statistics collected during the range collection phase of optimization, including simulation minimum and simulation maximum values. You can click on any result to view additional details in the **Result Details** pane. The **Visualization of Simulation Data** pane displays a summary of histograms of the bits used by each object in your model.

You can customize the information displayed in the **Results** spreadsheet, or use the **Explore** tab to sort and filter these results based on additional criteria. For more information, see “Control Views in the Fixed-Point Tool”.

The best solution found during optimization, **Solution 1**, is automatically applied to the model. To compare this optimized solution to the baseline run, click **Compare**. In the `Embedded Efficient Filter` subsystem, you can see the applied optimized fixed-point data types. When you click **Compare** for a model that has logged signals, the tool opens the **Simulation Data Inspector**. In the Simulation Data Inspector, select `output_signal` as the signal to compare. The plot of the plant output signal for **Solution 1** is within the specified tolerance band.



You can continue exploring other solutions by selecting a solution from the solutions table and clicking **Apply and Compare**.

After optimizing data types in the Fixed-Point Tool, you can choose to export the optimization workflow steps to a MATLAB® script. This allows you to save the current optimization workflow steps and continue data type optimization using `fxpopt` at the command line.

Click **Export Script** to export a script named `fxpOptimizationOptions` to the current working directory.

```
model = 'mSimpleFIR';

sud = 'mSimpleFIR/Embedded Efficient Filter';

options = fxpOptimizationOptions();

options.AdvancedOptions.UseDerivedRangeAnalysis = true; % Run range analysis as part of range collection.

addTolerance(options, 'mSimpleFIR/outut_signal', 1, 'AbsTol', 0.01);

addTolerance(options, 'mSimpleFIR/outut_signal', 1, 'RelTol', 0.01);

result = fxpopt(model, sud, options);

solution = explore(result);
```

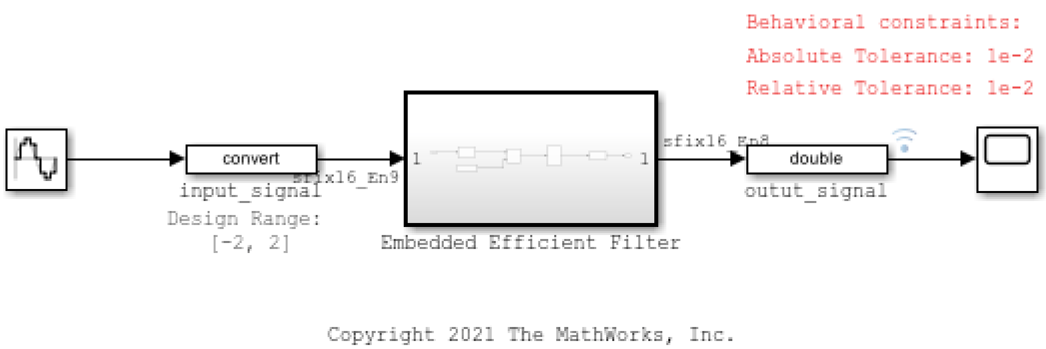
After the conversion process, if you want to restore your model to its state at the start of the conversion process, click **Restore Original Model**. Any changes made to your model after the preparation stage of conversion are removed.

Iterative Fixed-Point Conversion in the Fixed-Point Tool

This example shows how to use the **Iterative Fixed-Point Conversion** workflow in the **Fixed-Point Tool**. The model used in this example is a simple FIR filter modeled using initial guesses for fixed-point data types. In this example, you specify known behavioral constraints for the output of the filter and improve the fixed-point data types in the `Embedded Efficient Filter` subsystem.

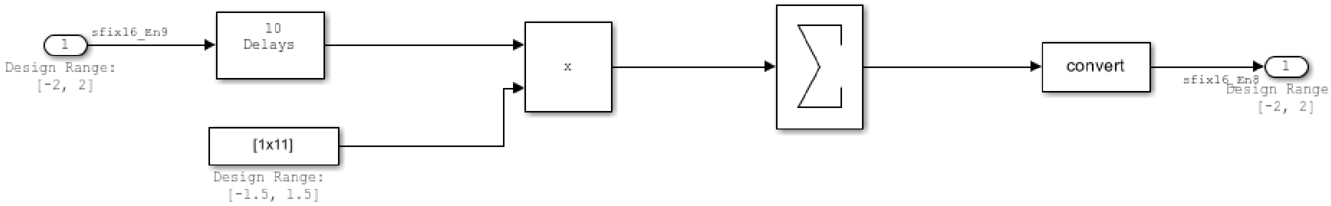
Open the `mSimpleFIR_fxp` model.

```
open_system('mSimpleFIR_fxp');
```



Inspect the `Embedded Efficient Filter` subsystem.

```
open_system('mSimpleFIR_fxp/Embedded Efficient Filter');
```



Known design minimum and maximum values are specified explicitly on blocks in the model, including on the inputs and outputs of the `Embedded Efficient Filter` subsystem.

Open the Fixed-Point Tool. On the Simulink® **Apps** tab, under **Code Generation**, click the app icon.

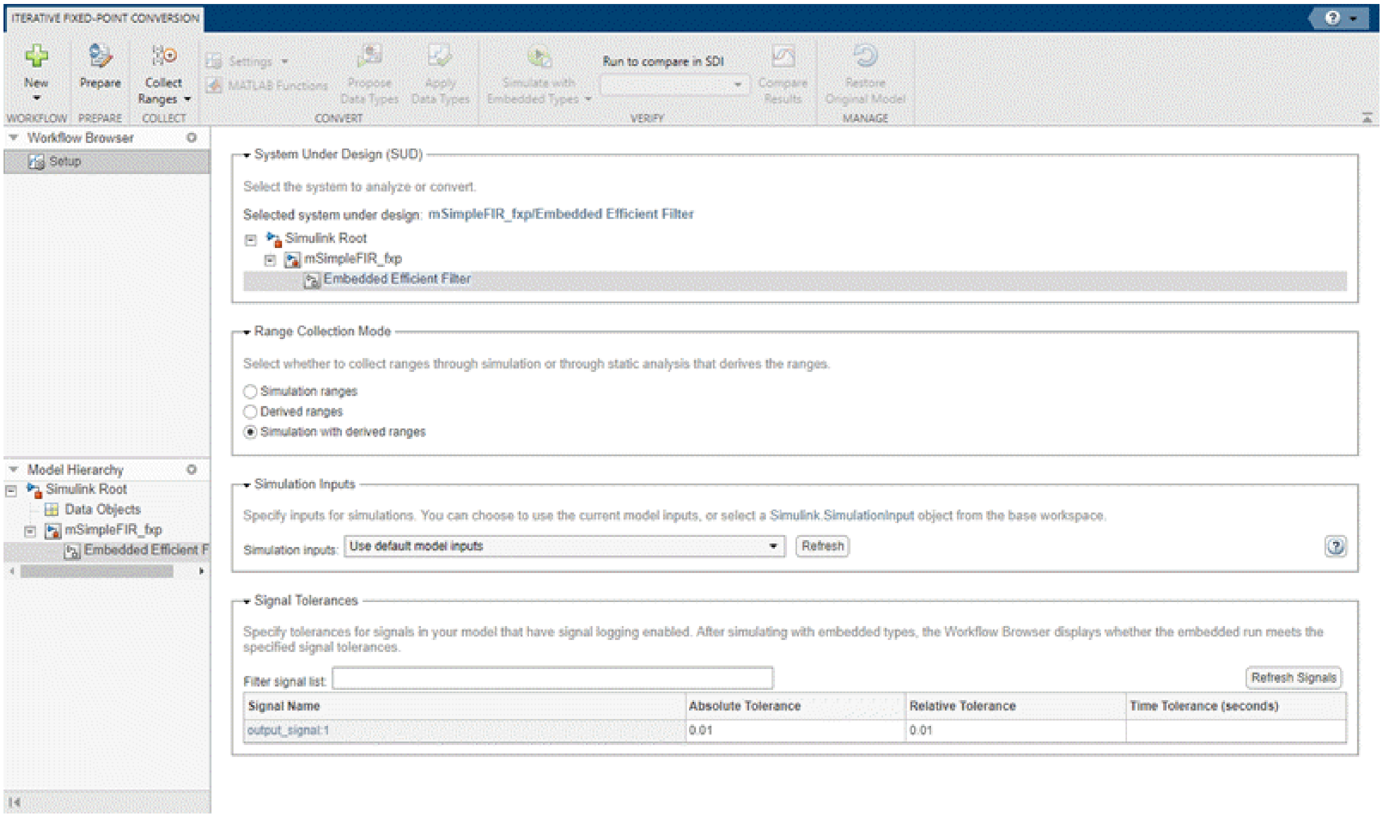
To start the iterative fixed-point conversion workflow, select **Iterative Fixed-Point Conversion**.

Select the subsystem that you want to analyze. Under **System Under Design (SUD)**, select the `Embedded Efficient Filter` subsystem.

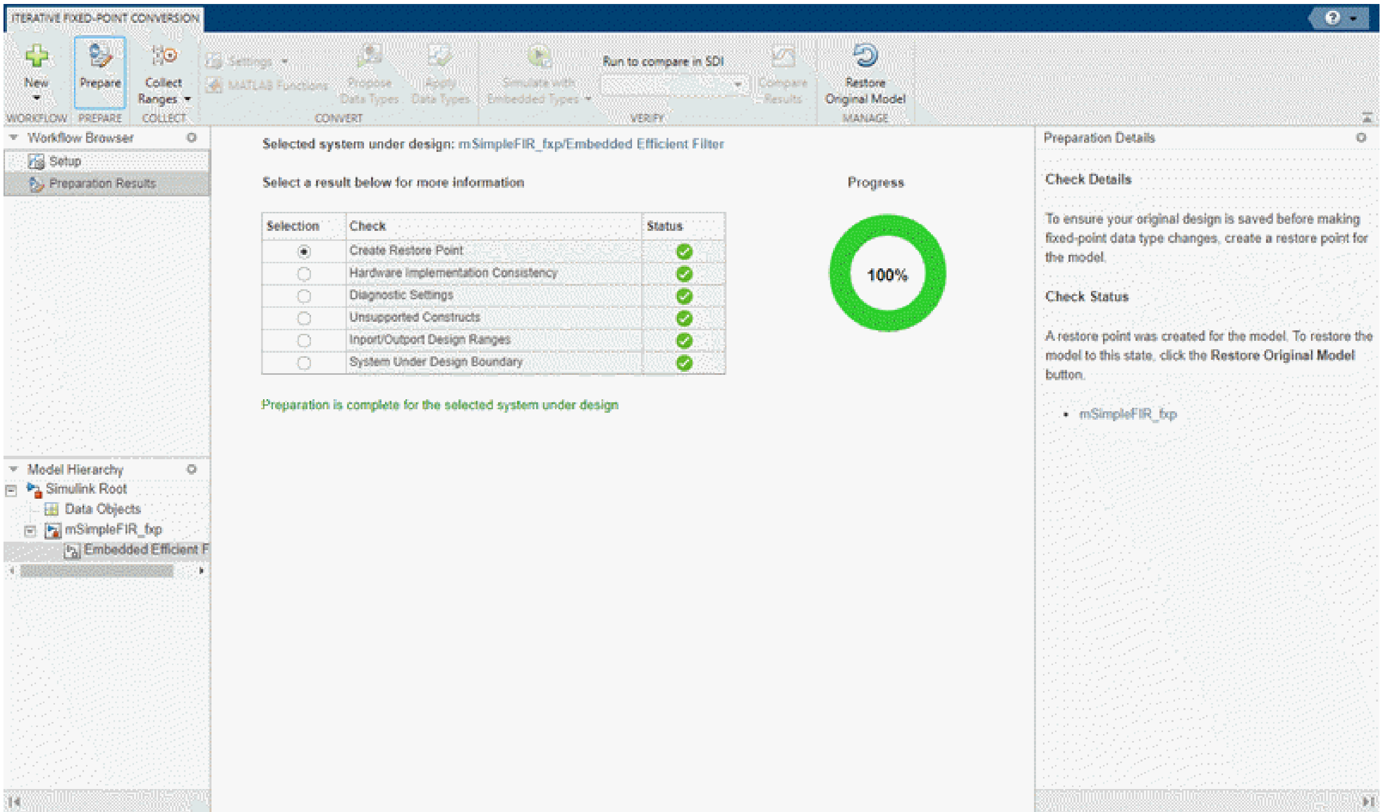
Choose the range collection method to use. Under **Range Collection Mode**, select **Simulation with derived ranges**. During the range analysis step of optimization, the tool will combine ranges from simulation minimum and maximum values, design minimum and maximum values specified explicitly on blocks in the model, and derived minimum and maximum values that are computed through a static analysis that derived ranges for objects in the model.

Specify **Simulation Inputs**. For this example, use the default model inputs for simulation.

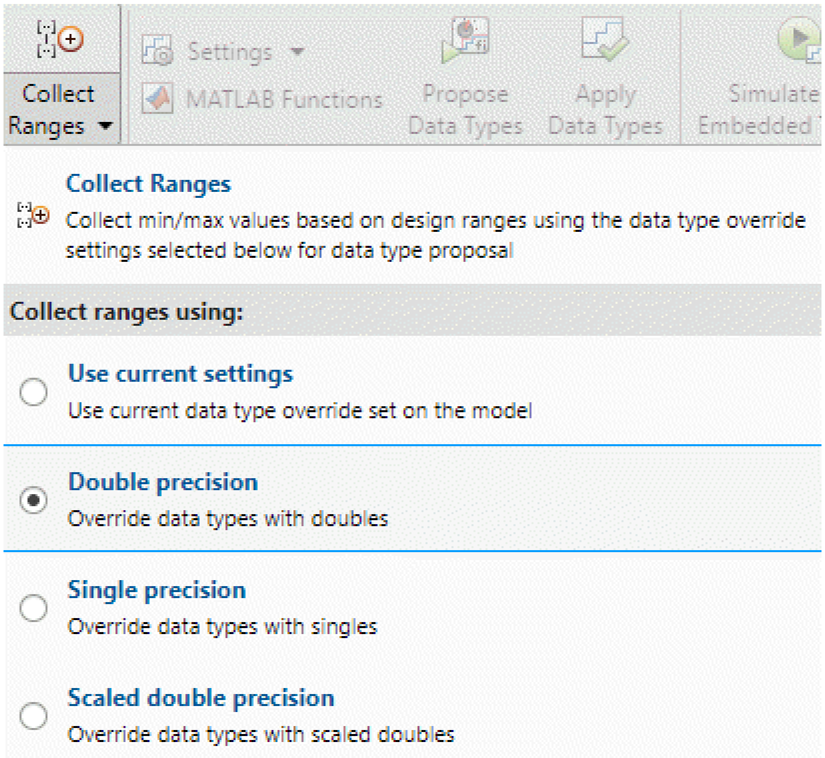
Specify signal tolerances for logged signals. Set the **Absolute Tolerance** and **Relative Tolerance** of the `output_signal:1` to `0.01`.



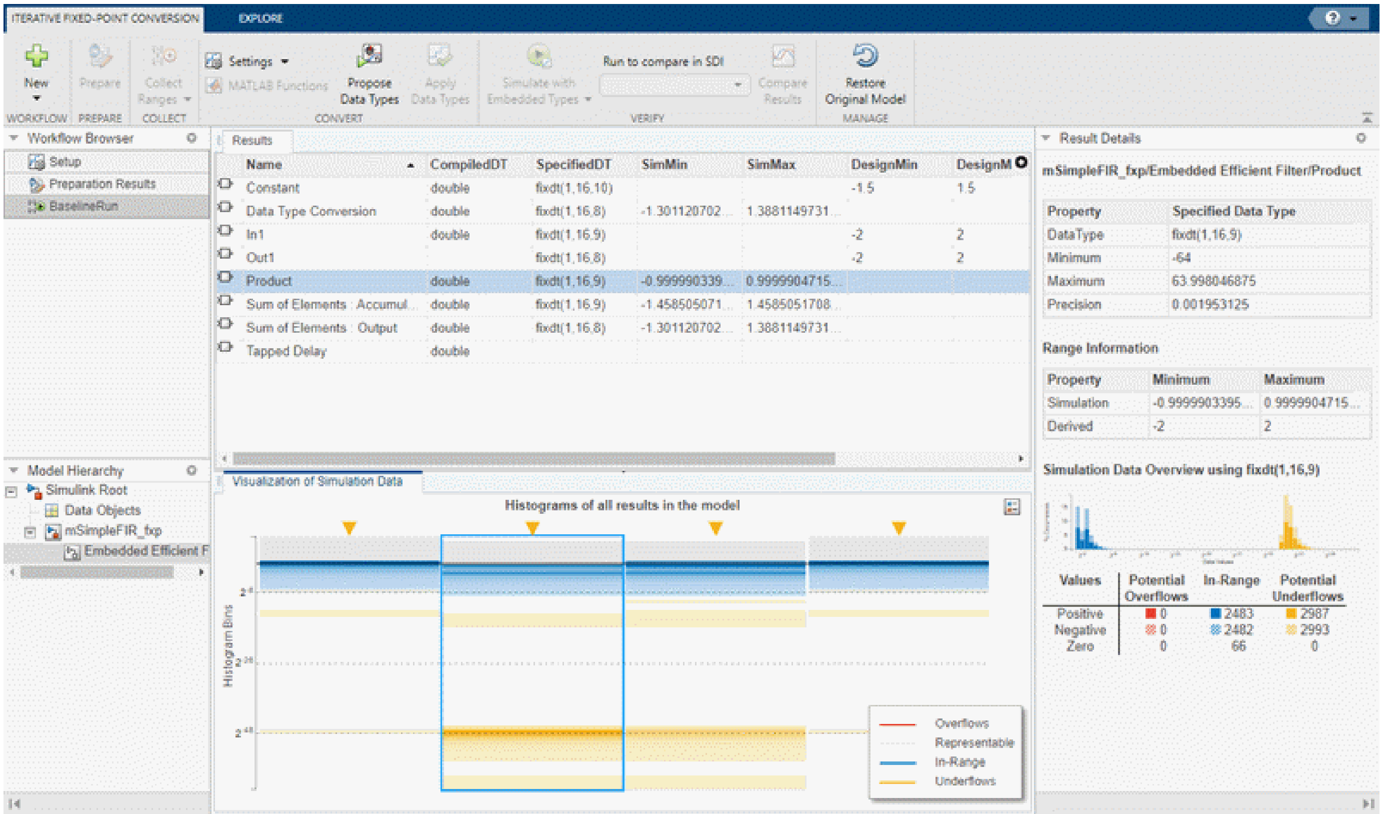
To prepare the model for fixed-point conversion, click **Prepare**. The Fixed-Point Tool creates a backup version of the model and checks the model for comaptibility with the conversion process. For more about preparation checks, see “Use the Fixed-Point Tool to Prepare a System for Conversion”.



Next, collect ranges. Expand the **Collect Ranges** button arrow and select **Double precision**. Click **Collect Ranges** to start the range collection run.



When you select **Double precision** as the range collection mode, the tool simulates the system under design with data type override enabled. Data type override performs a global override of the fixed-point data types in the model, thereby avoiding quantization effects. This enables you to establish an ideal floating-point baseline for the behavior of your model.

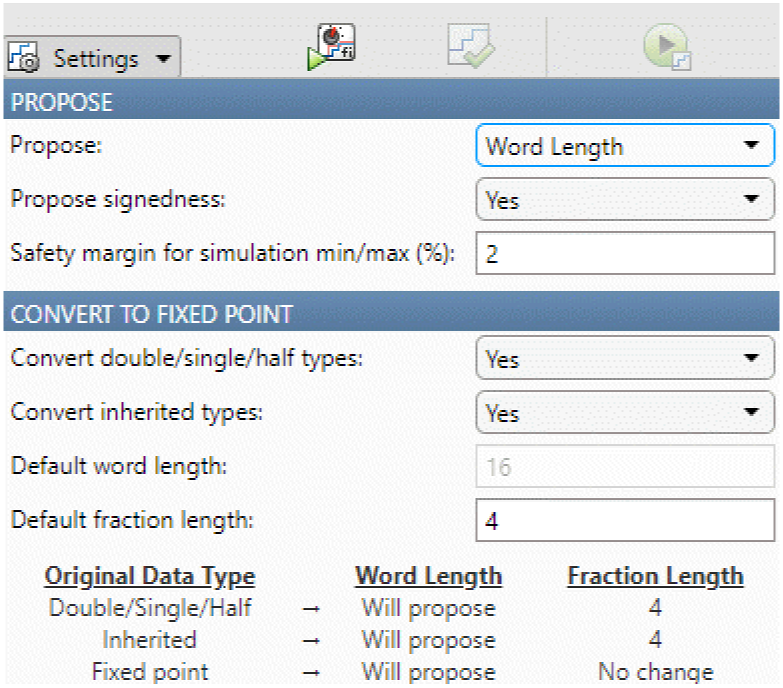


The results of range collection are stored in `BaselineRun`. The **Results** spreadsheet displays a summary of the statistics collected during the range collection simulation, including the currently specified data types on the model (**SpecifiedDT**), simulation minimum, and simulation maximum values. The compiled data type (**CompiledDT**) column displays double for all objects in the `Embedded Efficient Filter` subsystem, indicating that data type override was applied during the range collection simulation.

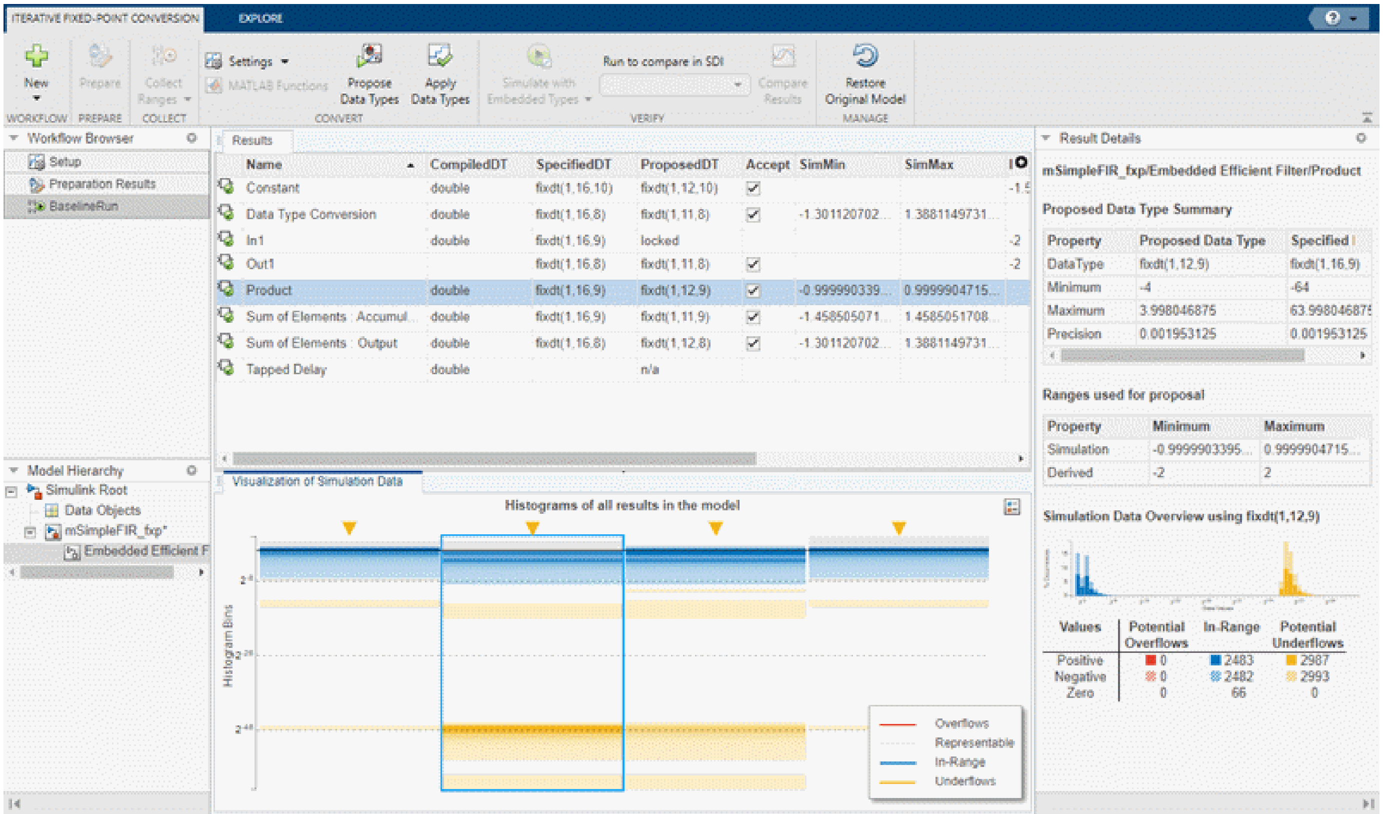
You can click on any result to view additional details in the **Result Details** pane. The **Visualization of Simulation Data** pane displays a summary of histograms of the bits used by each object in your model. The simulation data shows that several objects in the model have potential underflows.

You can customize the information displayed in the **Results** spreadsheet, or use the **Explore** tab to sort and filter these results based on additional criteria. For more information, see “Control Views in the Fixed-Point Tool”.

Next, expand the **Settings** button arrow to configure the settings to use for data type proposals. Set **Propose** to **Word Length**.



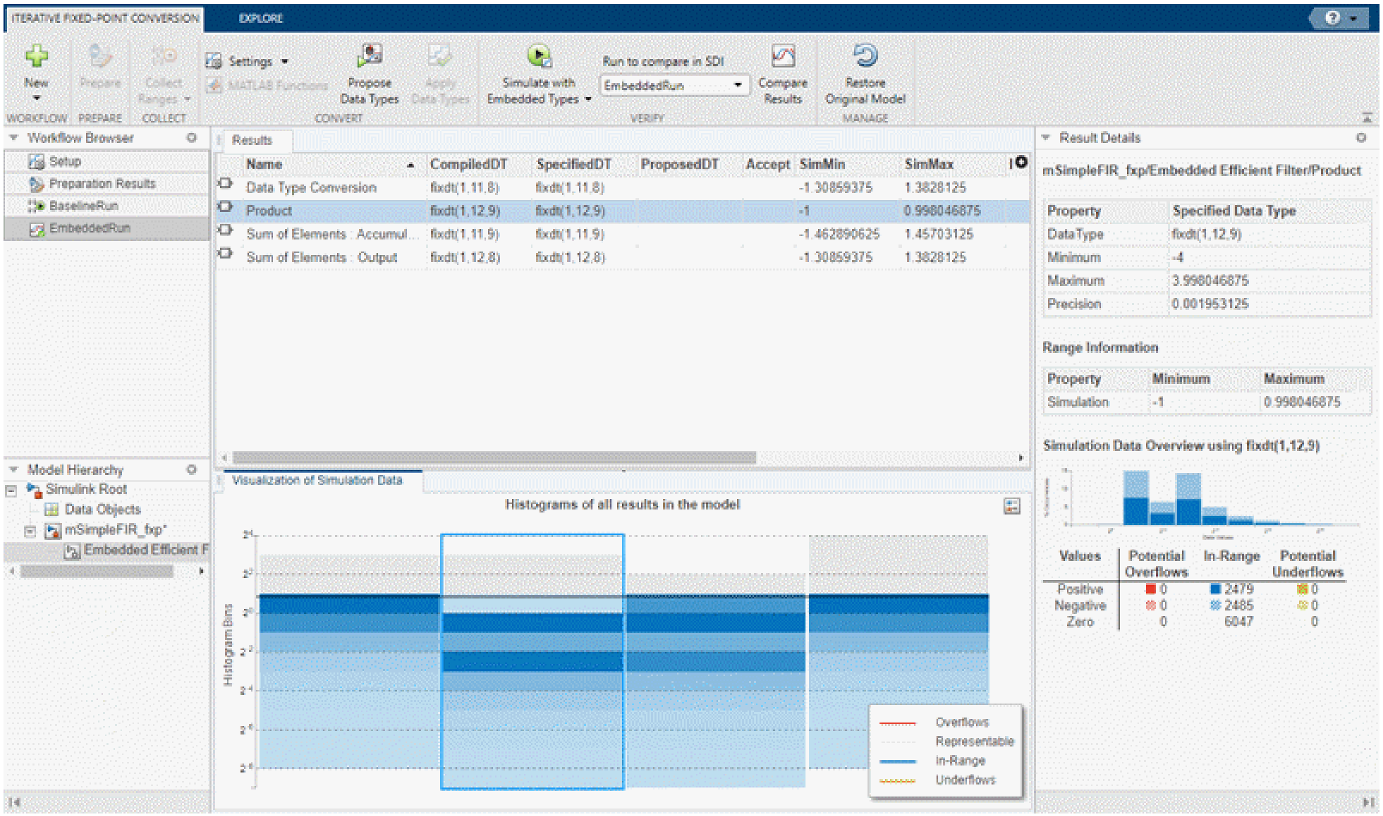
To propose data types based on the ranges collected and the data type proposal settings specified, click **Propose Data Types**. The tool uses all available range data to calculate data type proposals which can include design minimum or maximum values, simulation minimum or maximum values, and derived minimum or maximum values. Data types are proposed for all objects in the system under design whose **Lock output data type setting against changes by the fixed-point tools** parameter is cleared.



To write the proposed data types to the model, click **Apply Data Types**. The tool updates the **SpecifiedDT** column to show that the data types have been applied to the model.

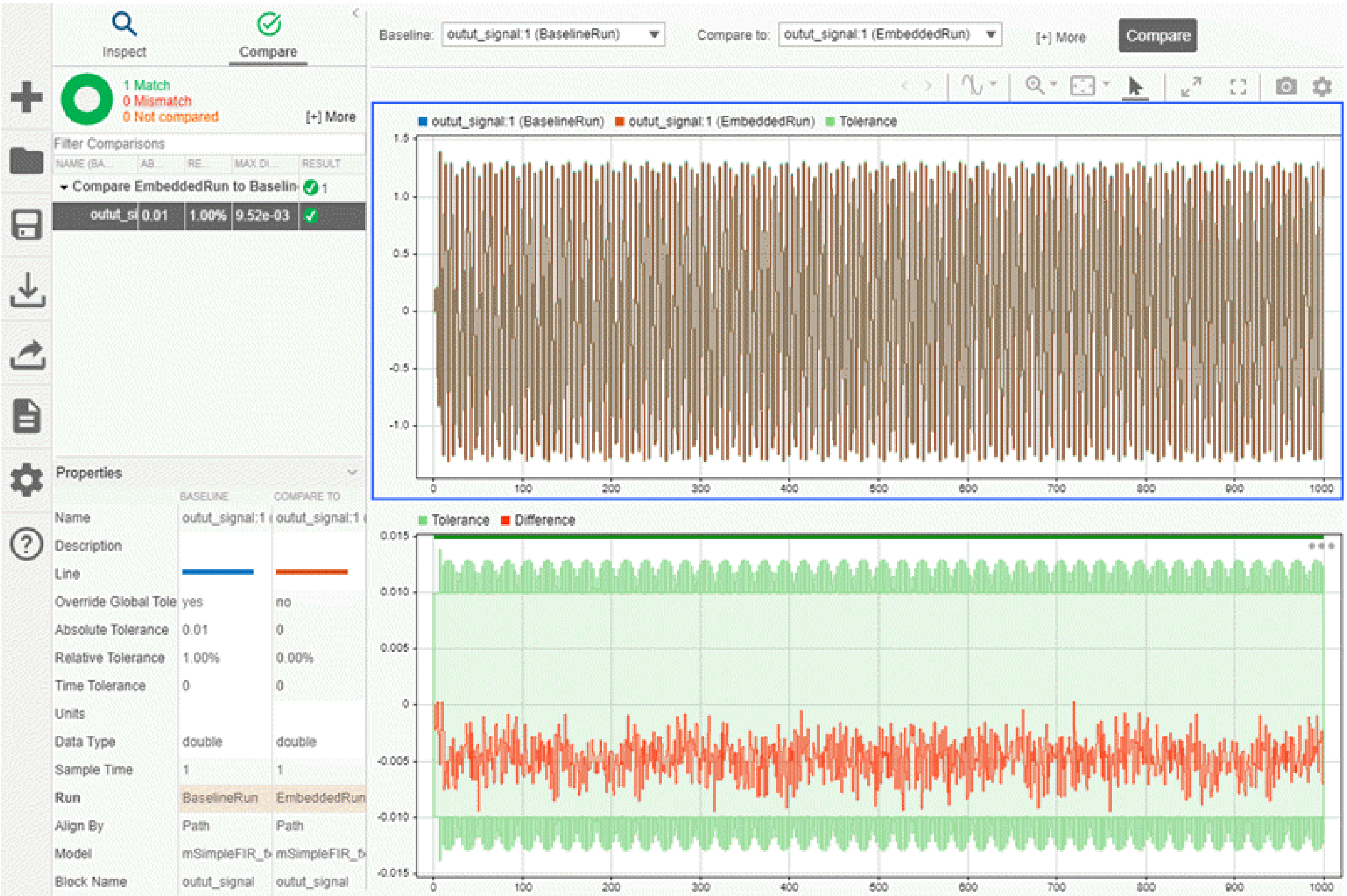
Simulate the model using the applied fixed-point data types. Expand the **Simulate with Embedded Types** button arrow and select **Specified data types**. Then click **Simulate with Embedded Types**.

The Fixed-Point Tool simulates the model using the new fixed-point data types and logs minimum and maximum values and overflow data for all objects in the system under design. This information is stored in a new run named `EmbeddedRun`. The icon next to `EmbeddedRun` displays a pass status, indicating that all signals in the system under design meet the specified tolerances. The **Visualization of Simulation Data** pane updates to display the new `EmbeddedRun` data.



To compare the ideal results stored in `BaselineRun` with the newly applied fixed-point data types, select `EmbeddedRun` from the **Run to compare in SDI** drop down menu. Then click **Compare Results** to open the **Simulation Data Inspector**.

In the Simulation Data Inspector, select `output_signal` as the signal to compare.



The plot of the filter output signal for `EmbeddedRun` is within the specified tolerance band.

If the behavior of the converted system does not meet your requirements or if you wish to explore the effect of additional data type selections, you can propose new data types after applying new proposal settings. Continue iterating until you find settings for which the fixed-point behavior of the system is acceptable.

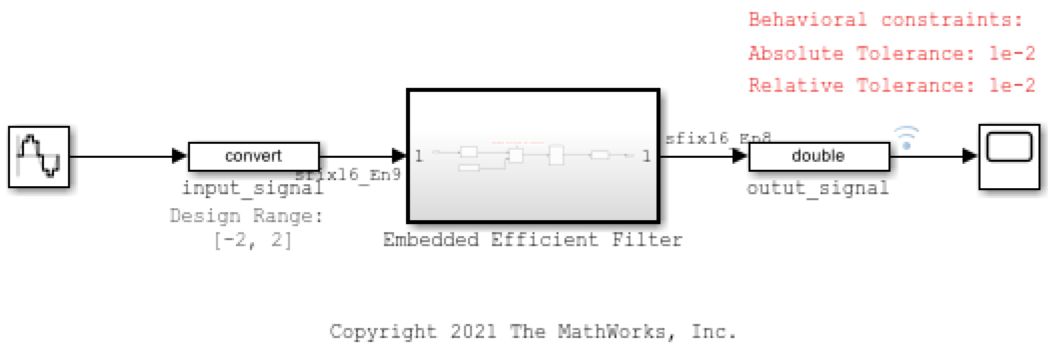
After the conversion process, if you want to restore your model to its state at the start of the conversion process, click **Restore Original Model**. Any changes made to your model after the preparation stage of conversion are removed.

Range Collection in the Fixed-Point Tool

This example shows how to use the **Range Collection** workflow in the **Fixed-Point Tool**. The model used in this example is a simple FIR filter modeled using fixed-point data types. In this example, you analyze the numerical behavior of the model to determine the source of overflow in the `Embedded Efficient Filter` subsystem.

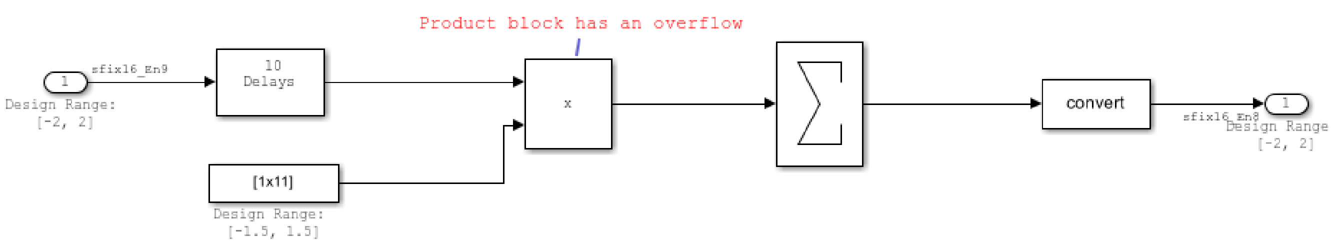
Open the `mSimpleFIR_fxp_ovf` model.

```
open_system('mSimpleFIR_fxp_ovf');
```



Inspect the `Embedded Efficient Filter` subsystem.

```
open_system('mSimpleFIR_fxp_ovf/Embedded Efficient Filter');
```



Known design minimum and maximum values are specified explicitly on blocks in the model, including on the inputs and outputs of the `Embedded Efficient Filter` subsystem.

Open the Fixed-Point Tool. On the Simulink® **Apps** tab, under **Code Generation**, click the app icon.

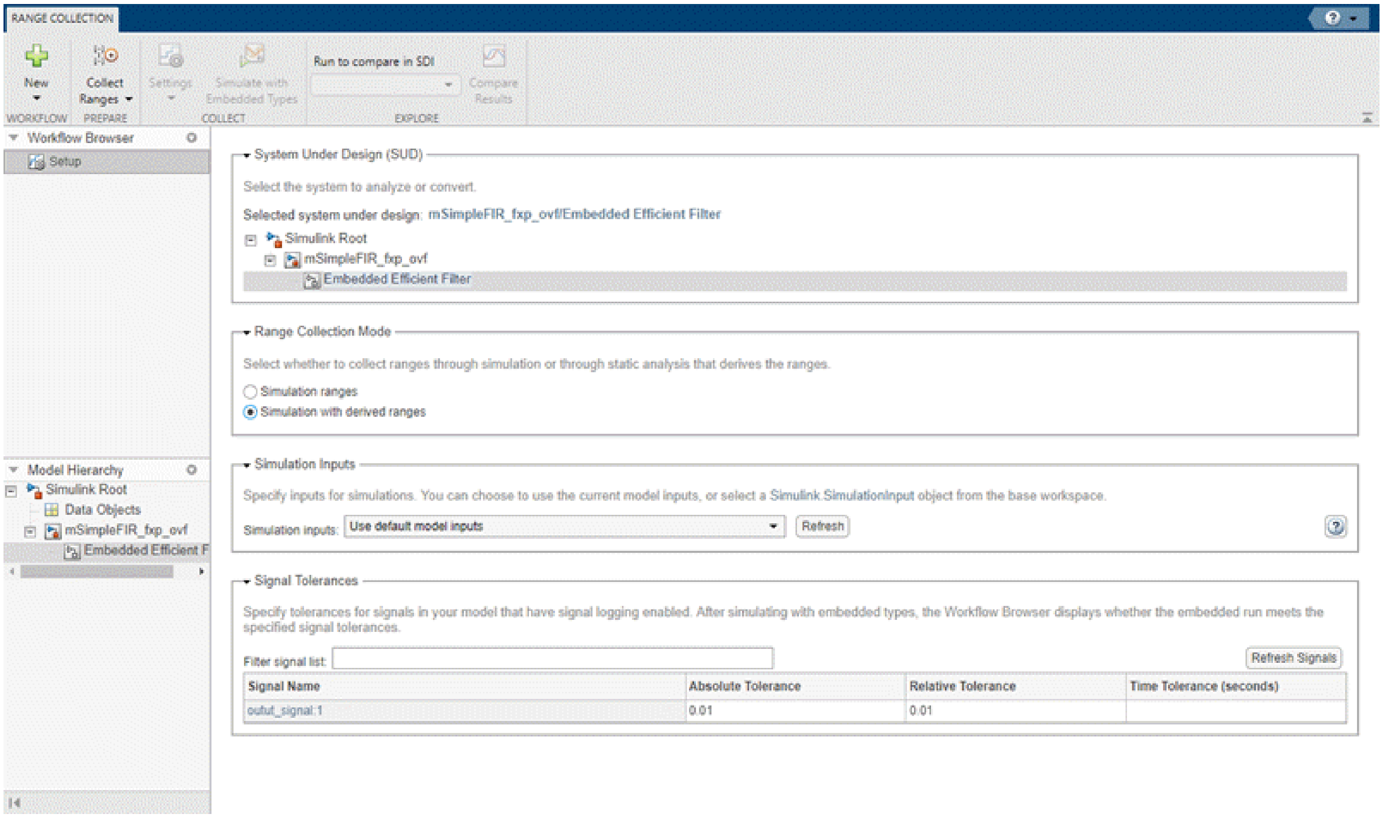
To start the range collection workflow, select **Range Collection**.

Select the subsystem that you want to analyze. Under **System Under Design (SUD)**, select the `Embedded Efficient Filter` subsystem.

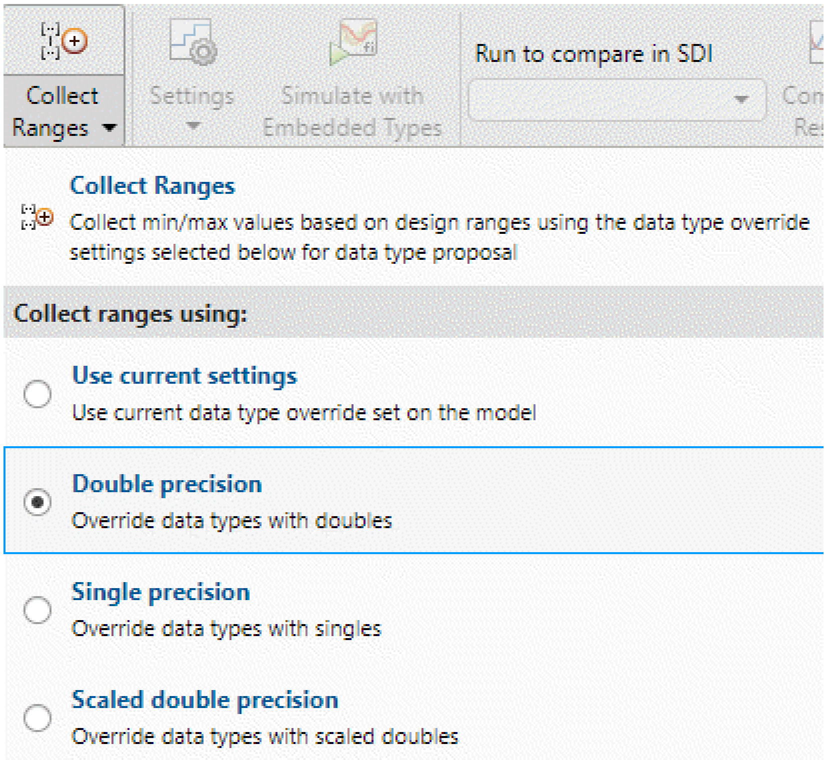
Choose the range collection method to use. Under **Range Collection Mode**, select **Simulation with derived ranges**. During range collection, the tool will combine ranges from simulation minimum and maximum values, design minimum and maximum values specified explicitly on blocks in the model, and derived minimum and maximum values that are computed through a static analysis that derived ranges for objects in the model.

Specify **Simulation Inputs**. For this example, use the default model inputs for simulation.

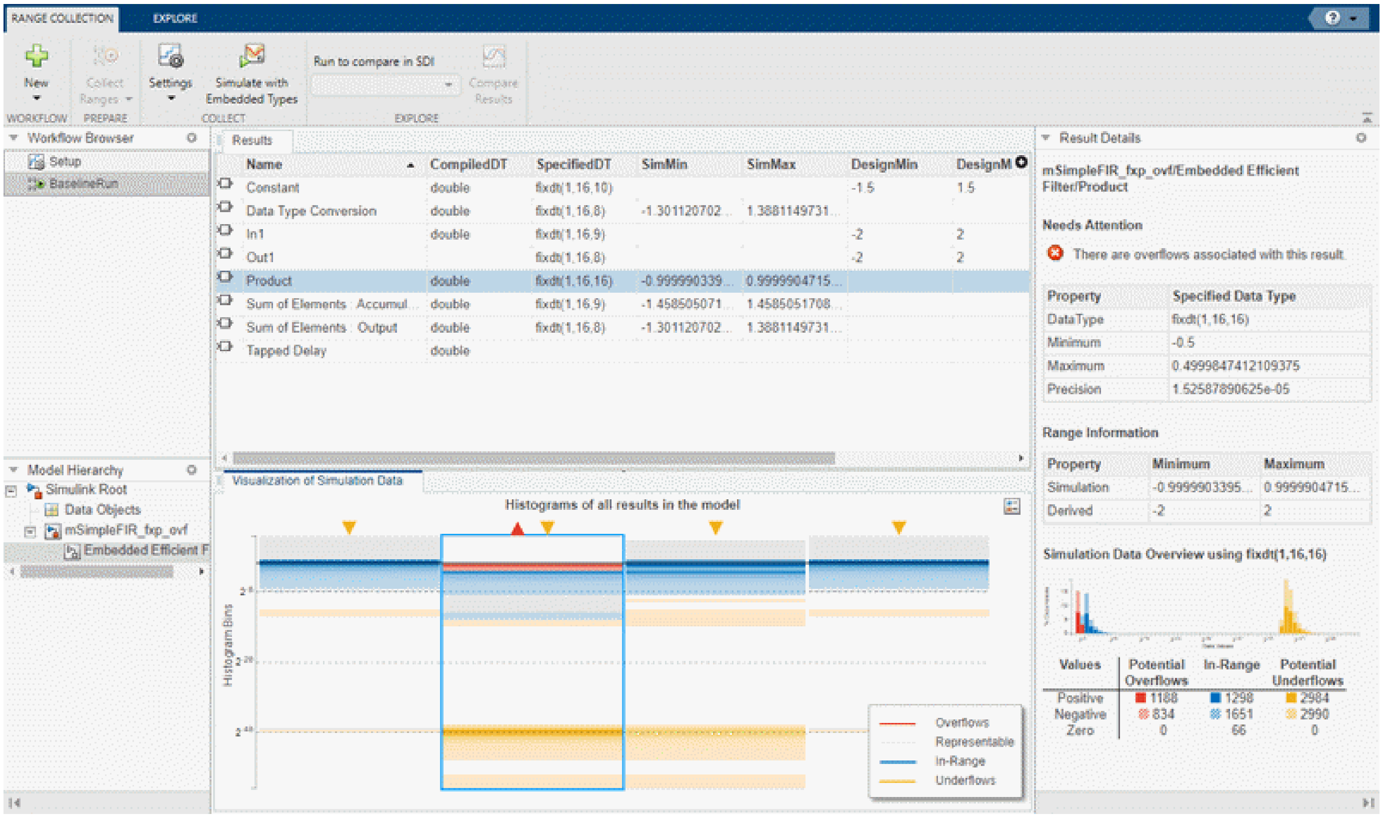
Specify signal tolerances for logged signals. Set the **Absolute Tolerance** and **Relative Tolerance** of the `output_signal:1` to `0.01`.



Next, expand the **Collect Ranges** button arrow to configure the settings to use for range collection. Select **Double precision** to temporarily override data types in the model with doubles during the baseline range collection run. Click **Collect Ranges**.



The results of the range collection run are stored in `BaselineRun`. The **Results** spreadsheet displays a summary of the statistics collected during the range collection, including the currently specified data types on the model (**SpecifiedDT**), simulation minimum, and simulation maximum values. The compiled data type (**CompiledDT**) column displays double for all objects in the `Embedded Efficient Filter` subsystem, indicating that data type override was applied during the range collection simulation.

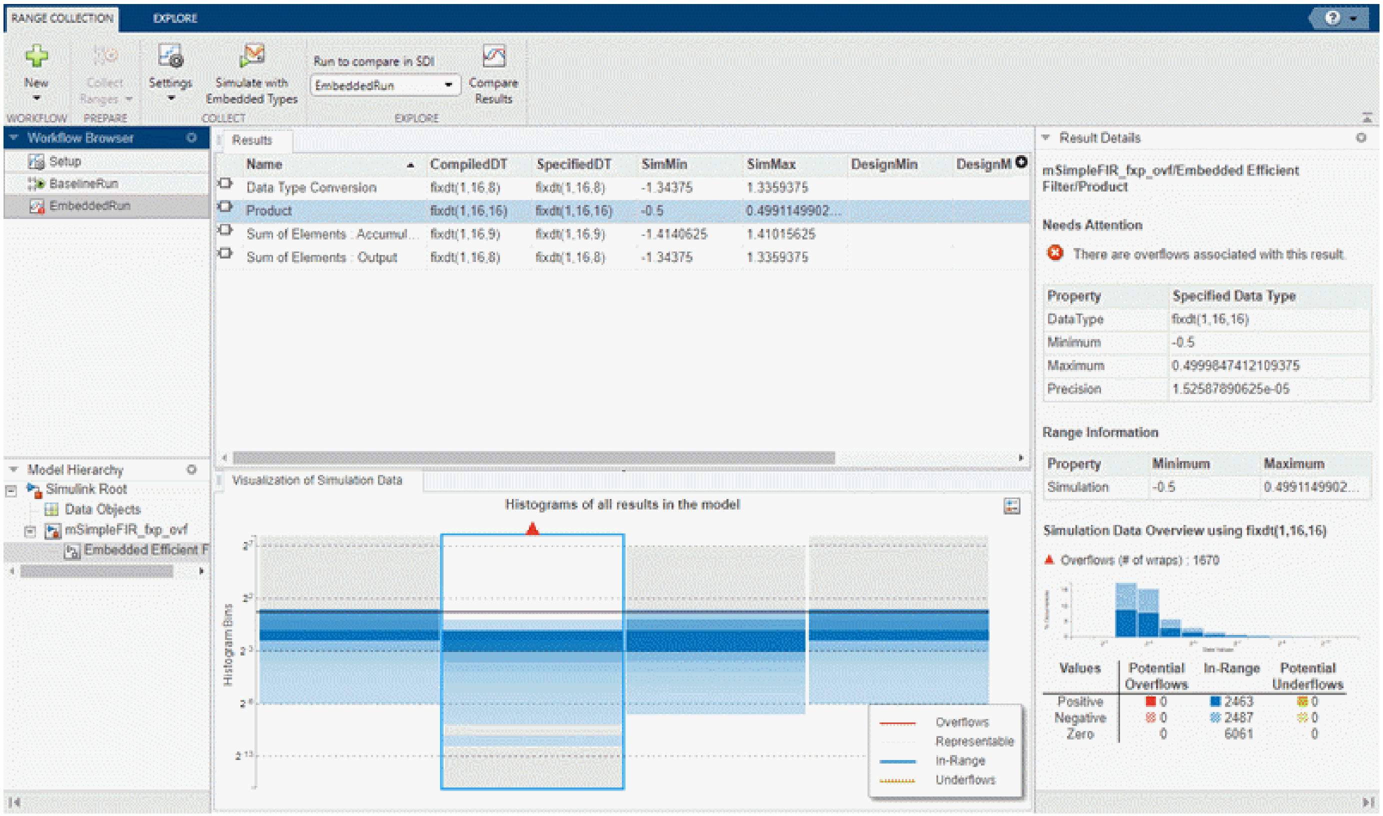


You can click on any result to view additional details in the **Result Details** pane. The **Visualization of Simulation Data** pane displays a summary of histograms of the bits used by each object in your model.

You can customize the information displayed in the **Results** spreadsheet, or use the **Explore** tab to sort and filter these results based on additional criteria. For more information, see “Control Views in the Fixed-Point Tool”.

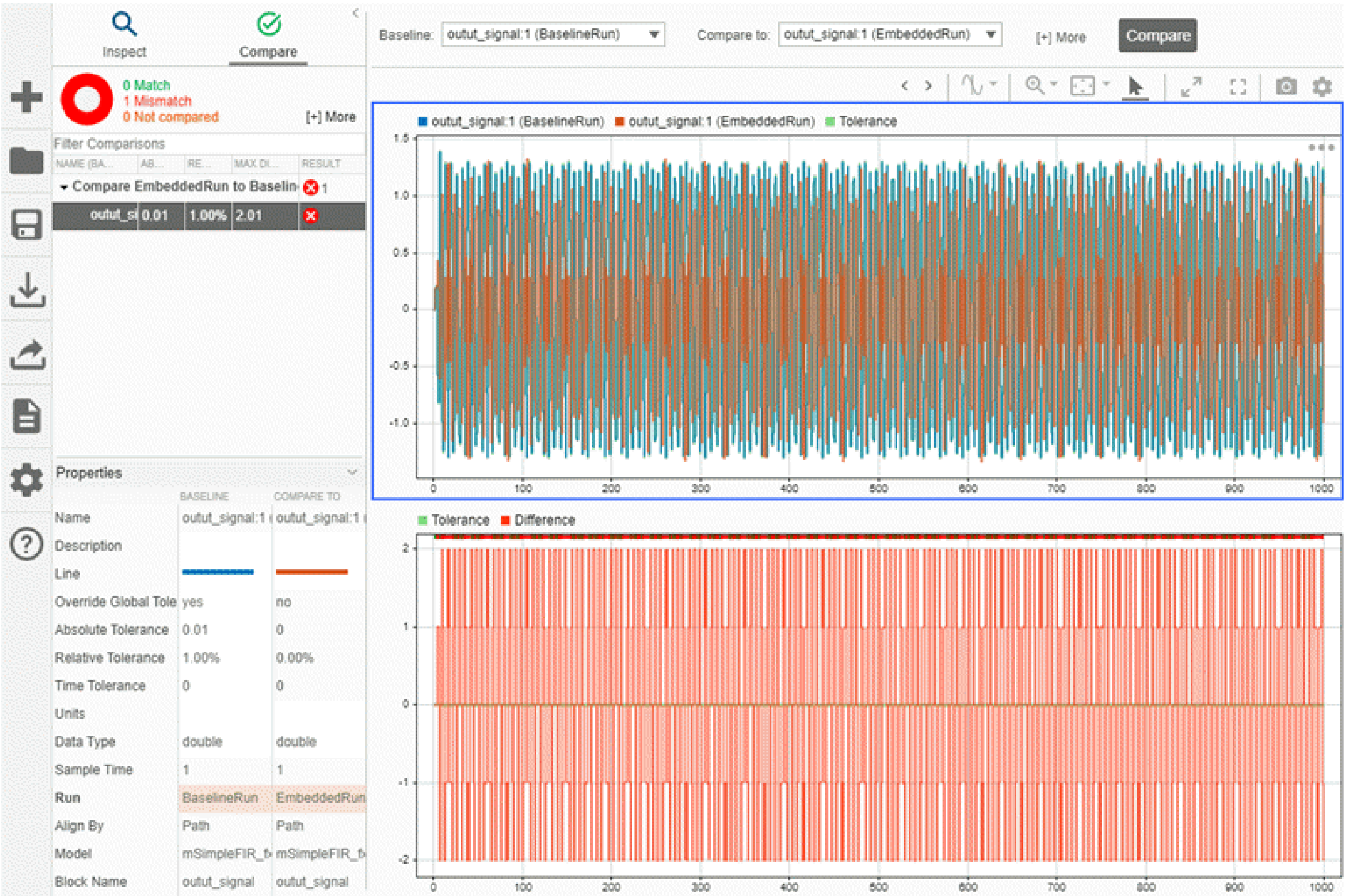
Next, simulate the model using the fixed-point data types currently specified on the model. Expand the **Settings** button arrow and select **Specified data types**, then click **Simulate with Embedded Types**.

The Fixed-Point Tool stores the results of the simulation in `EmbeddedRun`.



The icon next to EmbeddedRun displays a fail status, indicating that one or more signals do not meet the specified tolerances. The results for the Product block indicate that there is an issue with this result. The **Result Details** pane shows that the block overflowed 1670 times, indicating a poor choice of word length.

To compare the ideal results stored in BaselineRun with the fixed-point results, select EmbeddedRun from the **Run to compare in SDI** drop down menu. Then click **Compare Results** to open the **Simulation Data Inspector**. In the Simulation Data Inspector, select output_signal as the signal to compare.



- “Convert Floating-Point Model to Fixed Point”
- “Optimize the Fixed-Point Data Types of a System Using the Fixed-Point Tool”
- “Perform Data Type Optimization with Custom Behavioral Constraints”
- “Use the Fixed-Point Tool to Explore Numerical Behavior”

Parameters

`System Under Design (SUD)` — System or subsystem to analyze or convert
current system (default)

System or subsystem to analyze or convert to fixed-point. You can select individual subsystems in your model one at a time to facilitate debugging by isolating the source of numerical issues, or you can choose the top-level model.

For more information on converting systems containing particular modeling constructs, see:

- “Convert a Referenced Model to Fixed Point”
- “Bus Objects in the Fixed-Point Workflow”
- “Autoscaling Data Objects Using the Fixed-Point Tool”

- “Convert MATLAB Function Block to Fixed Point”

`Range Collection Mode` — How the tool collects ranges for objects in your system
Simulation ranges (default) | **Derived ranges** | **Simulation with derived ranges**

How the tool collects ranges for objects in your system, specified as one of the following:

- **Simulation ranges** — Collect ranges through simulation. To collect and merge the ranges of multiple simulation runs, specify “Simulation Inputs” on page 1-0 . Data type proposals are as good as the test bench provided.
- **Derived ranges** — Collect ranges through a static analysis that derives the ranges, also known as *range analysis* or *derived range analysis*. Ranges collected using this option are based only on design ranges specified on the model. This option typically delivers more conservative data type proposals. For more information, see “How Range Analysis Works”.
- **Simulation with derived ranges** — Collect ranges through simulation and derived range analysis and combine the results. Proposed data types are based on the union of simulation and derived ranges. This option provides the most comprehensive range information.

For more information, see “Choosing a Range Collection Method”.

`Simulation Inputs` — Inputs for simulations
`Use default model inputs` (default) | `Simulink.SimulationInput` object

Inputs for simulations, specified as a `Simulink.SimulationInput` object.

If you choose the “Range Collection Mode” on page 1-0 to be **Simulation ranges** or **Simulation with derived ranges**, you can choose to specify additional simulation inputs to improve the accuracy of the collected ranges and data type proposals. During the range collection simulation, the **Fixed-Point Tool** captures the minimum and maximum values from each specified simulation scenario. If the `Simulink.SimulationInput` object that you select contains more than one simulation scenario, the **Fixed-Point Tool** proposes data types based on the merged ranges from all simulation scenarios.

A comprehensive set of input signals that exercise the full range of your design will result in more accurate data type proposals for your system. For an example, see “Propose Data Types For Merged Simulation Ranges”.

`Signal Tolerances` — Tolerances for signals in your model that have signal logging enabled
absolute tolerance | relative tolerance | time tolerance

To determine if the numerical behavior of a new fixed-point implementation is acceptable, you can define tolerances for individual signals in your model that have logging enabled. You can specify any of the following types of tolerances:

- **Absolute Tolerance** — Absolute value of the maximum acceptable difference between the original signal and the signal in the converted design.
- **Relative Tolerance** — Maximum relative difference, specified as a percentage, between the original output and the output of the new design. For example, a value of `1e-2` indicates a maximum difference of one percent between the original values and the signal values of the converted design.
- **Time Tolerance (seconds)** — Time interval in which the maximum and minimum values define the upper and lower values to compare against.

In the **Optimized Fixed-Point Conversion** workflow, you must specify at least one behavioral constraint in order to optimize data types. Signal tolerances are one type of behavioral constraint that you can specify.

In the **Iterative Fixed-Point Conversion** workflow, signal tolerances are not required to propose data types, but are required for the tool to determine whether the embedded run is within tolerance.

In the **Range Collection** workflow, signal tolerances are not required to collect ranges, but are required for the tool to determine whether the ranges collected are within tolerance.

You can enter signal tolerances using any valid MATLAB expression that returns a finite, non-negative value.

For more information, see "Specify Behavioral Constraints" and "Tolerance Computation".

`Collect Ranges` — Collect ranges
`Use current settings` (default) | `Double precision` | `Single precision` | `Scaled double precision`

Collect ranges for objects in your model using:

- `Use current settings` — Use the current data type override set on the model.
- `Double precision` — Override data types in the model with doubles.
- `Single precision` — Override data types in the model with singles.
- `Scaled double precision` — Override data types in the model with scaled doubles.

Ranges collected depend on the "Range Collection Mode" on page 1-0 and any "Simulation Inputs" on page 1-0 specified.

For more information, see "Fixed-Point Instrumentation and Data Type Override" and "Use Custom Data Type Override Settings for Range Collection".

`Settings` — Data typing options
`Allowable Wordlengths` | `Max Iterations` | `Propose` | `Propose signedness` | `Verify using` | ...

Data typing options available in the **Settings** menu depend on the workflow chosen.

Optimized Fixed-Point Conversion Workflow Options

Option	Description
`Allowable Wordlengths`	`[2:128]` (default) Word lengths that can be used in your optimized system under design. The final result of the optimization uses word lengths in the intersection of the `Allowable Wordlengths` and word lengths compatible with hardware constraints specified in the **Hardware Implementation** pane of your model.

Option	Description
`Max Iterations`	`50` (default) Maximum number of iterations to perform, specified as a scalar integer. The optimization process iterates through different solutions until it finds an ideal solution, reaches the maximum number of iterations, or reaches another stopping criteria.
`Max Time (sec)`	`600` (default) Maximum amount of time for the optimization to run, specified in seconds as a scalar number. The optimization runs until it reaches the time specified, an ideal solution, or another stopping criteria.
`Patience (iterations)`	`10` (default) Maximum number of iterations where no new best solution is found, specified as a scalar integer. The optimization continues as long as the algorithm continues to find new best solutions.
`Safety Margin (%)`	`0` (default) A safety margin, specified as a positive scalar value, indicating the percentage increase in the bounds of the collected range. The safety margin is applied to the union of all collected ranges.

Option	Description
`Objective Function`	Objective function to use during the optimization search. The optimization algorithm seeks to minimize an objective function while meeting the specified behavioral constraints. • `Bit Width Sum` (default) — Minimize total bit width sum. • `Operator Count` — Minimize estimated count of operators in generated C code. This option may result in a lower program memory size for C code generated from Simulink models. The `'OperatorCount'` objective function is not suitable for FPGA or ASIC targets. **Note** To use `Operator Count` as the objective function during optimization, the model must be ready for code generation. For more information about determining code generation readiness, see “Check Model and Configuration for Code Generation” (Embedded Coder).
`Perform Neighborhood Search`	`on` (default) Whether to perform a neighborhood search for the optimized solution. Disabling this option can increase the speed of the optimization process, but also increases the chances of finding a less ideal solution.
`Use Parallel`	`off` (default) Whether to run iterations of the optimization in parallel. Running the iterations in parallel requires a Parallel Computing Toolbox™ license. If you do not have a Parallel Computing Toolbox license, or if you do no enable this option, the iterations run in serial.

Iterative Fixed-Point Conversion Workflow Options

Option	Description
Propose	Whether to propose fraction lengths or word lengths for objects in the system under design. • Fraction Length (default) — The **Fixed-Point Tool** uses range information and the specified Default word length value to propose best-precision fraction lengths for the objects in your model. • Word Length — The **Fixed-Point Tool** uses range information and the specified Default fraction length value to propose word lengths for the objects in your model.
Propose signedness	Yes (default) Whether to use the collected range information to propose signedness.
Safety margin for simulation min/max (%)	2 (default) Specify a safety margin to apply to collected simulation ranges. The **Fixed-Point Tool** will add the specified amount to the collected ranges and base proposals on this larger range.
Convert double/single/half types	Yes (default) Whether to generate data type proposals for objects that currently specify a double, single, or half-precision data type.
Convert inherited types	Yes (default) Whether to generate data type proposals for results that currently specify an inherited data type.
Default word length	16 (default) Default word length to use for data type proposals, specified as a scalar integer. This setting is enabled only when the Propose setting is set to Fraction Length.
Default fraction length	4 (default) Default fraction length to use for data type proposals, specified as a scalar integer. This setting is enabled only when the Propose setting is set to Word Length.

Range Collection Workflow Options

Option	Description
`Verify using`	Data type override settings to use for embedded simulation. • `Specified data types` — Use data types specified on the model • `Scaled double precision` — Override data types with scaled doubles.

Limitations

- Some blocks do not support fixed-point data types and can result in an error during fixed-point conversion. See “Blocks That Do Not Support Fixed-Point Data Types”.
- Some modeling constructs may cause data type propagation issues. See “Models That Might Cause Data Type Propagation Errors”.
- If your model contains a MATLAB Function block, use only supported modeling constructs for successful conversion. See “MATLAB Language Features Supported for Automated Fixed-Point Conversion”.

Tips

- For best practices and recommendations, see “Best Practices for Fixed-Point Conversion Workflow”.
- To customize views in the **Fixed-Point Tool**, see “Control Views in the Fixed-Point Tool”.
- For help troubleshooting the optimization workflow, see “Data Type Optimization Not Successful”.

Version History

Introduced before R2006a

See Also

`fxptdlg` | `DataTypeWorkflow.Converter` | `fxpopt` | “Optimize Fixed-Point Data Types for a System” | “The Command-Line Interface for the Fixed-Point Tool”

Topics

“Convert Floating-Point Model to Fixed Point”
“Optimize the Fixed-Point Data Types of a System Using the Fixed-Point Tool”
“Perform Data Type Optimization with Custom Behavioral Constraints”
“Use the Fixed-Point Tool to Explore Numerical Behavior”

Lookup Table Optimizer

Optimize existing lookup table or approximate function with lookup table

Description

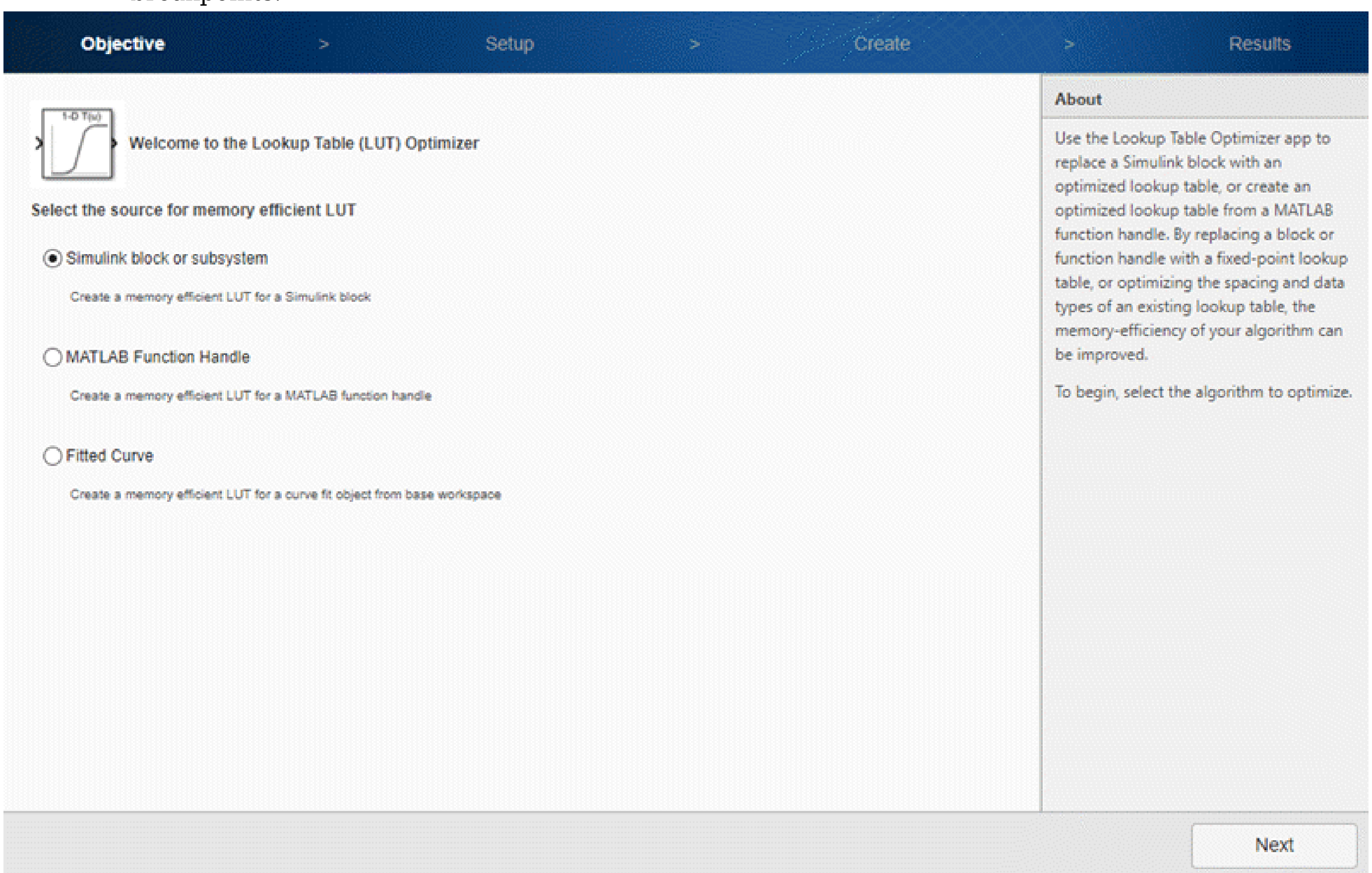
Use the **Lookup Table Optimizer** app to obtain an optimized (memory-efficient) lookup table.

Using this app, you can:

- Approximate an existing Simulink block, including Subsystem blocks and math function blocks
- Approximate a MATLAB handle
- Approximate a curve fit object

You can choose to return the optimized lookup table as a Simulink block or as a MATLAB function.

The optimizer supports any combination of floating-point and fixed-point data types. The original input and output data types can be kept or changed as desired. To minimize memory used, the optimizer selects the data types of breakpoints and table data as well as the number and spacing of breakpoints.



Open the Lookup Table Optimizer App

- Simulink toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
- In a Simulink model with a Lookup Table block, select the Lookup Table block. In the **Lookup Table** tab, select **Lookup Table Optimizer**.

Examples

- "Optimize Lookup Tables for Memory-Efficiency"
- "Generate an Optimized Lookup Table as a MATLAB Function"
- "Convert Floating-Point Model to Fixed Point"

Parameters

`Source` — Source for memory-efficient LUT
Simulink block or subsystem (default) | MATLAB function handle | curve fit object

Select the source for memory-efficient LUT:

- Simulink block or subsystem — Simulink block or subsystem to approximate, or lookup table block to optimize, for example, 1-D Lookup Table or n-D Lookup Table. If you specify one of the lookup table blocks, the app generates an optimized lookup table.
- MATLAB function handle — MATLAB function handle to approximate with a lookup table. Function handles must be on the MATLAB search path, or approximation fails.
- Fitted curve — Curve fit `cfit` object from the base workspace to approximate. For a list of library models to approximate, see "List of Library Models for Curve and Surface Fitting" (Curve Fitting Toolbox).

Tip The process of generating a lookup table approximation is faster for a function handle than for a subsystem. If a subsystem can be represented by a function handle, approximating the function handle is faster.

`Output Error Tolerance` — Tolerance of difference between original and approximation
non-negative scalar

Specify the maximum tolerance of the `Absolute` and `Relative` difference between the original output value and the output value of the approximation.

`Allowed Word Lengths` — Word lengths permitted in lookup table approximation
`[8 16 32]` (default) | vector of integers

Specify the word lengths, in bits, that can be used in the lookup table approximation based on your intended hardware. For example, if you intend to target an embedded processor, you can restrict the data types in your lookup table to native types. The word lengths must be between 1 and 128.

`LUT Specification` — Options for optimized lookup table
`Interpolation` | `Breakpoint specification` | `Saturate to output type` | `AUTOSAR Compliant` | `Explore Half` | `HDL Optimized` | `Solution Type`

Specify options to use for the optimized lookup table.

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td><code>Interpolation</code></td><td>When an input falls between breakpoint values, the lookup table interpolates the output value using neighboring breakpoints.
<code>Linear</code> — Fits a line between the adjacent breakpoints and returns the point on that line that corresponds to the input.
<code>Flat</code> — Returns the output value that corresponds to the breakpoint value that is immediately less than the input value. If no breakpoint value exists below the input value, this option returns the breakpoint value nearest to the input value.
<code>Nearest</code> — Returns the value that corresponds to the breakpoint that is closest to the input. If the input is equidistant from two adjacent breakpoints, this option selects the breakpoint with the higher index.
<code>None</code> — Generates a Direct Lookup Table (n-D) block, which performs table lookups without any interpolation or extrapolation.

Note When generating a Direct Lookup Table block, the maximum number of inputs is two.</td></tr>
<tr><td><code>Breakpoint specification</code></td><td>Spacing of breakpoint data.
<code>ExplicitValues</code> — Lookup table breakpoints are specified explicitly. Breakpoints can be closer together for some input ranges and farther apart in others.
<code>EvenSpacing</code> — Lookup table breakpoints are evenly spaced throughout.
<code>EvanPow2Spacing</code> — Lookup table breakpoints use power-of-two spacing. This breakpoint specification boasts the fastest execution speed because a bit shift can replace the position search.
For more information on how breakpoint specification can affect performance, see “Effects of Spacing on Speed, Error, and Memory Usage”.</td></tr>
<tr><td><code>Saturate to output type</code></td><td>Whether to automatically saturate the range of the output of the function to approximate to the range of the output data type.</td></tr>
</table>

Option	Description
`AUTOSAR Compliant`	Whether the generated lookup table is AUTOSAR compliant. When this option is set to `True`, • The generated lookup table is a Curve or Map block from the AUTOSAR Blockset. • The data type of the table data must equal the output data type of the block. • An AUTOSAR Blockset license is checked out. This option is not supported when the `Solution Type` option is set to `MATLAB`.
`Explore Half`	Whether to allow the optimizer to explore half-precision data types for table data and breakpoints.
`HDL Optimized`	Whether to generate an HDL-optimized approximation. When this option is set to `True`, the generated approximation is a subsystem consisting of a prelookup step followed by interpolation that functions as a lookup table with explicit pipelining to generate efficient HDL code. To generate an HDL-optimized approximation, the function to approximate must be one-dimensional and `Breakpoint specification` must be set to `EvenSpacing` or `EvenPow2Spacing`. This property is not supported when the `Solution Type` option is set to `MATLAB`.

Option	Description

Solution Type	How the app outputs the optimized lookup table.
	• Simlink — Generate a Simulink subsystem containing the optimized lookup table.
	• MATLAB — Output the optimized lookup table as a MATLAB function. Generating an optimized lookup table as a MATLAB function is not supported when:
	  • The AUTOSARCompliant property is set to true.
	  • The UseParallel property is set to true.
	  • The HDLOptimized property is set to true.
	  • The InterpolationMethod property is set to 'None'.
	Note The Simulink block and MATLAB function lookup table approximations generated by the **Lookup Table Optimizer** may not be exactly numerically equivalent. However, both solution forms are guaranteed to meet all constraints specified in the optimization problem.

Settings — Optimization options
Max time | Max memory usage (bytes) | On curve table values | Use parallel

Specify additional optimization options.

Option	Description
Max time	Maximum amount of time for the approximation to run, specified in seconds as a scalar number. The approximation runs until it reaches the time specified, finds an ideal solution, or reaches another stopping criterion. Default: Inf
Max memory usage (bytes)	The maximum amount of memory the generated lookup table can use, in bytes, specified as a scalar integer. Default: 10000000

Option	Description
`On curve table values`	Whether to constrain table values to the quantized output of the function being approximated. When you set this option to `False` and allow off-curve table values, you may be able to reduce the memory of the lookup table while maintaining the same error tolerances or maintain the same memory while reducing the error tolerances. Default: `False`
`Use parallel`	Whether to run iterations of the optimization in parallel. Running iterations in parallel requires a Parallel Computing Toolbox license. If you do not have a Parallel Computing Toolbox, or if you specify `False`, the iterations run in serial. This option is not supported when the `Solution Type` option is set to `MATLAB`. Default: `False`

Limitations

- Lookup table objects and breakpoint objects are not supported in a model mask workspace.
- Functions and function handles that you approximate must meet these criteria:
  - The function must be time-invariant.
  - The function must operate element-wise, meaning for each input there is one output.
  - The function must not contain states.

  For more information, see “Vectorization”.

Algorithms

Infinite Upper and Lower Input Bounds

When object to approximate specifies infinite input ranges and the input type is non-floating-point, the software infers upper and lower ranges during the approximation based on the range of the input data type. The resulting lookup table solution specifies the bounds that the algorithm used during the approximation, not the originally specified infinite bounds.

Upper and Lower Input Bounds and Input Data Type Range

If the `Minimum` or `Maximum` specified for an input fall outside the range of the specified `Desired Data Type`, the algorithm uses the range of the data type specified by `Desired Data Type` for the approximation.

In cases where the `Breakpoint specification` option is set to `EvenSpacing` but the specified `Minimum` or `Maximum` values of the input is equal to the range of the `Desired Data Type`, the algorithm does not attempt to find a solution using `EvenPow2Spacing`.

Version History

Introduced in R2018a

R2022b: Support for curve fitting objects

The **Lookup Table Optimizer** app now supports curve fitting `cfit` objects as valid inputs for approximation.

R2022a: Improved memory reduction for 1-D and flat interpolation

The **Lookup Table Optimizer** has an improved algorithm for lookup table value and breakpoint optimization for one-dimensional functions with flat interpolation. This enhancement can enable improved memory reduction of the optimized lookup table and faster completion of the lookup table optimization process.

This improvement applies when the function to approximate is one-dimensional and all of these options are specified:

- `Interpolation` is set to `Flat`.
- `Breakpoint specification` is set to `ExplicitValues`.
- `On curve table values` is set to `False`.

R2021b: Generate an optimized lookup table approximation as a MATLAB function

Use the **Lookup Table Optimizer** app to generate an optimized lookup table approximation as a MATLAB function. The generated MATLAB function is editable and supports C/C++ code generation using MATLAB Coder.

R2021a: Generate optimized one-dimensional lookup tables for HDL applications

Use the **Lookup Table Optimizer** app to generate a subsystem consisting of a prelookup step followed by interpolation that functions as a lookup table with explicit pipelining to generate efficient HDL code.

R2021a: Lookup table optimization support for functions with scalar inputs

Previously, the **Lookup Table Optimizer** required that functions and function handles to approximate were vectorized, meaning that for each input, there is exactly one output. Starting in R2021a, lookup table optimization fully supports approximation of Simulink blocks and subsystems that only allow scalar inputs.

R2021a: Improved lookup table value optimization

The **Lookup Table Optimizer** has an improved algorithm for lookup table value optimization for the `Flat` and `Nearest` interpolation methods when off-curve table values are allowed. This enhancement can enable faster completion of the lookup table optimization process and improved memory reduction of the optimized lookup table.

R2021a: Stop optimization in Lookup Table Optimizer app

You can now stop the optimization solver in the **Lookup Table Optimizer** before the optimization search is complete. The app will choose the best solution found at the time the **Stop** button is clicked and display the solution in the app.

R2020b: Explore half precision in optimized lookup tables

Specify whether the optimization process explores half-precision data types for table data and breakpoint values.

R2020a: Iteratively redesign lookup tables in your model

The **Lookup Table Optimizer** now replaces blocks being approximated by a lookup table with a variant subsystem containing the function approximation. The variant subsystem enables you to return to the original function and perform the optimization again using different optimization settings and constraints.

R2020a: Parallelized lookup table optimization

Specify whether to run iterations of the optimization in parallel. Running iterations in parallel requires a Parallel Computing Toolbox license. If you do not have a Parallel Computing Toolbox license, or if you specify `False`, the iterations run in serial.

R2019b: Allow off-curve table values in optimized lookup tables

You can now generate an optimized lookup table with off-curve table values.

In previous releases, the optimization required table values to match the quantized output values of the original function being approximated. By allowing off-curve table values, you may be able to reduce the memory of the lookup table while maintaining the same error tolerances, or maintain the same memory while reducing the error tolerances.

R2019b: Generate optimized AUTOSAR-compliant lookup table

Generate an AUTOSAR-compliant optimized lookup table using a Curve or Map block.

Setting this option to `True` checks out an AUTOSAR Blockset license.

See Also

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTSolution` | `FunctionApproximation.LUTMemoryUsageCalculator` | `LUTCompressionResult`

Functions
`solve` | `approximate` | `compare` | `totalmemoryusage` | `solutionfromID` | `displayfeasiblesolutions` | `displayallsolutions`

Topics
"Optimize Lookup Tables for Memory-Efficiency"
"Generate an Optimized Lookup Table as a MATLAB Function"
"Convert Floating-Point Model to Fixed Point"

Single Precision Converter

Convert double-precision system to single precision

Description

The **Single Precision Converter** automatically converts a double-precision system to single precision.

During the conversion process, the converter replaces all user-specified double-precision data types, as well as output data types that compile to double precision, with single-precision data types. The converter does not change built-in integer, Boolean, or fixed-point data types.

Open the Single Precision Converter

- From the Simulink **Apps** tab, select **Single Precision Converter**.

Examples

- “Convert a System to Single Precision”

Programmatic Use

`report = DataTypeWorkflow.Single.convertToSingle(systemToConvert)` converts the system specified by `systemToConvert` to single-precision and returns a report. The `systemToConvert` must be open before you begin the conversion.

Version History

Introduced in R2016b

See Also

Functions
`convertToSingle`

Topics
“Convert a System to Single Precision”
“Getting Started with Single Precision Converter”

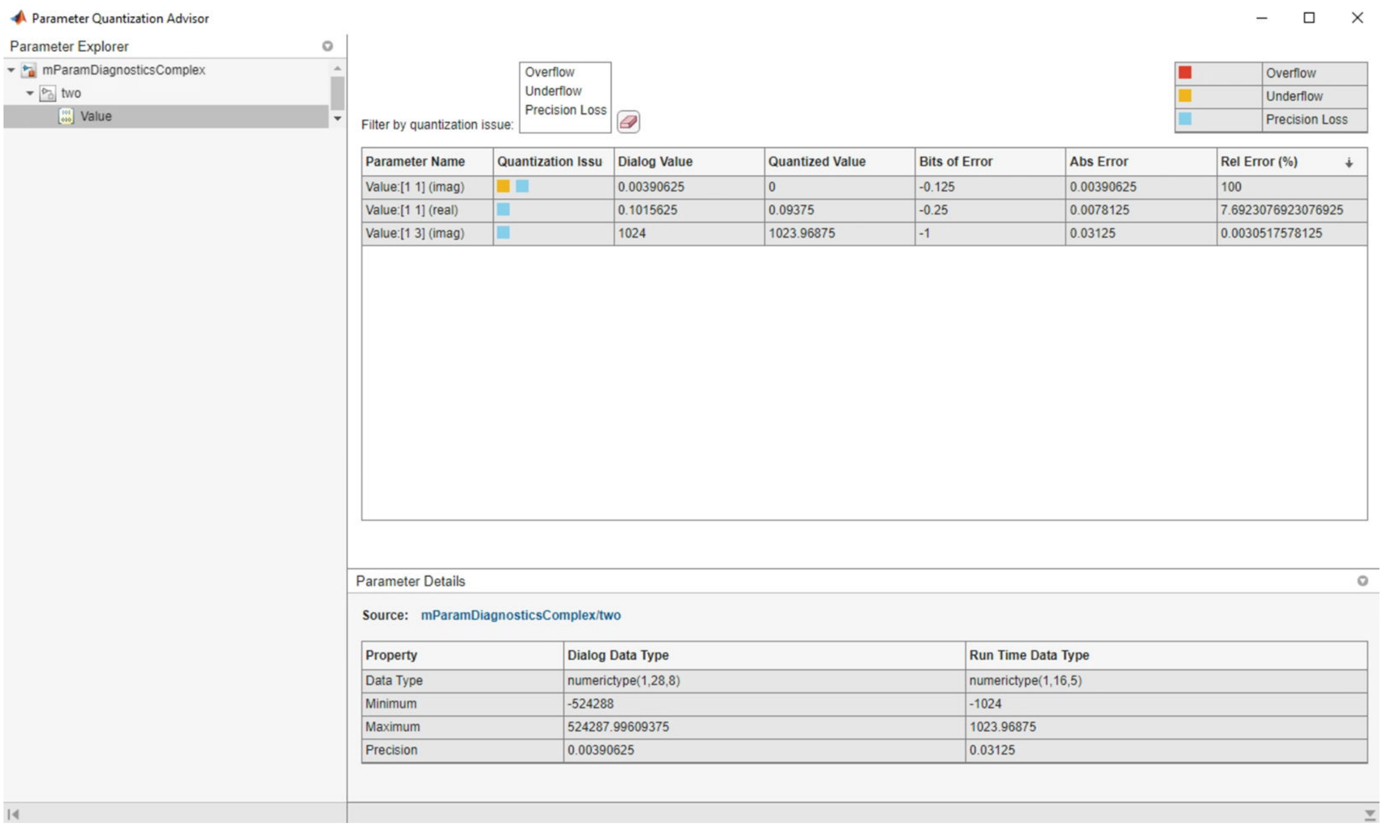
Parameter Quantization Advisor

Inspect numerical issues related to parameter quantization

Description

The **Parameter Quantization Advisor** app provides details on numerical issues related to parameter quantization. Using this app, you can:

- Filter quantization issues resulting from parameter overflow, underflow, and precision loss.
- Sort quantization issues based on bits of error, absolute error, or relative error.
- Explore in detail the effects of parameter quantization on your model.
- Explore all parameters associated with a model.



Open the Parameter Quantization Advisor App

- **Diagnostic Viewer**: From a numeric diagnostic warning or error, under **Suggested Actions**, click **Open**.

Examples

Explore Parameter Precision Loss

Use the **Parameter Quantization Advisor** to explore parameter precision loss in a feedback controller model.

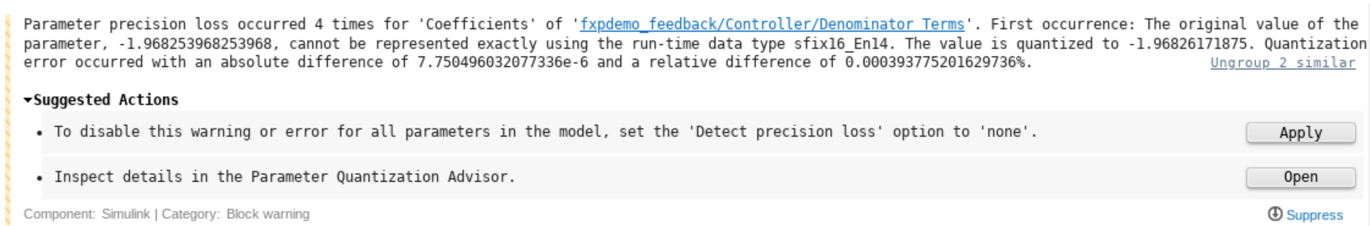
Open the `fxpdemo_feedback` model.

```
fxpdemo_feedback
```

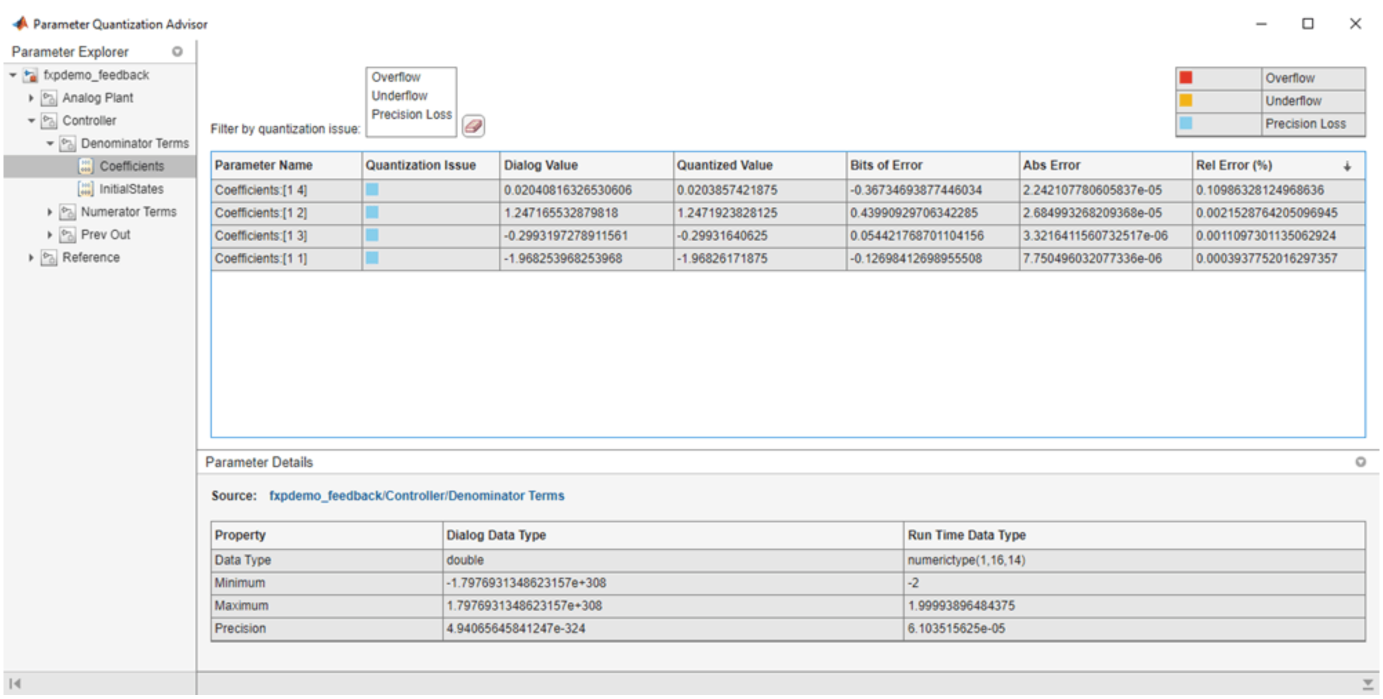
Check that parameter diagnostics are enabled. In the **Configuration Parameters** dialog, under **Diagnostics > Data Validity**, set the **Detect precision loss** parameter diagnostic to `warning`.

Simulate the model.

The **Diagnostic Viewer** displays a warning for parameter precision loss.



To inspect details of this diagnostic in the **Parameter Quantization Advisor**, from **Suggested Actions**, click **Open**.



The app displays details of the parameter precision loss that occurred in `fxpdemo_feedback/Denominator Terms`. In this example, the model has only precision loss issues, as indicated by a blue square in the **Quantization Issue** column of the table. You can also use the app to explore overflow and underflow issues.

Four coefficients in the Discrete FIR Filter block named `Denominator Terms` experienced precision loss. The table shows the dialog value entered for each parameter and the corresponding quantized value of each parameter. You can sort the table by bits of error, absolute error, or relative error.

The **Parameter Details** pane contains additional details for the parameter experiencing numeric issues, including the location in the model and the name, dimension, and complexity of the parameter. The table displays the data type of the parameter in the dialog and the data type of the parameter at run time along with the minimum, maximum, and precision representable by these data types.

Tips

- Update the diagram or simulate the model to view messages in the **Diagnostic Viewer**.
- The model must successfully compile in order to launch the **Parameter Quantization Advisor** app. If a diagnostic errors, the model will not compile and the app cannot be launched.
- The **Parameter Quantization Advisor** app reports information for these quantization issues:

  - "Detect underflow"
  - "Detect overflow"
  - "Detect precision loss"

  Diagnostics set to `none` are also reported in the app.
- The **Parameter Quantization Advisor** reports details on quantization issues for tunable and nontunable parameters that experience quantization loss.
- You can inspect all parameters associated with a model in the **Parameter Quantization Advisor**. A hierarchical tree of all parameters in the model is displayed in the **Parameter Explorer**. Select the parameter you would like to inspect to view additional details.

Version History

Introduced in R2022b

R2023a: View Multiple Parameters

The **Parameter Quantization Advisor** now allows you to explore all parameters associated with a model.

Additionally, the **Parameter Quantization Advisor** has expanded support for the MATLAB Function block, Stateflow® chart parameters, the FIR Decimation block, and the FIR Interpolation block.

See Also

"Detect underflow" | "Detect overflow" | "Detect precision loss"

Simulation Data Inspector

Inspect and compare data and simulation results to validate and iterate model designs

Description

The Simulation Data Inspector visualizes and compares multiple kinds of data.

Using the Simulation Data Inspector, you can inspect and compare time series data at multiple stages of your workflow. This example workflow shows how the Simulation Data Inspector supports all stages of the design cycle:

1. "View Data in the Simulation Data Inspector"

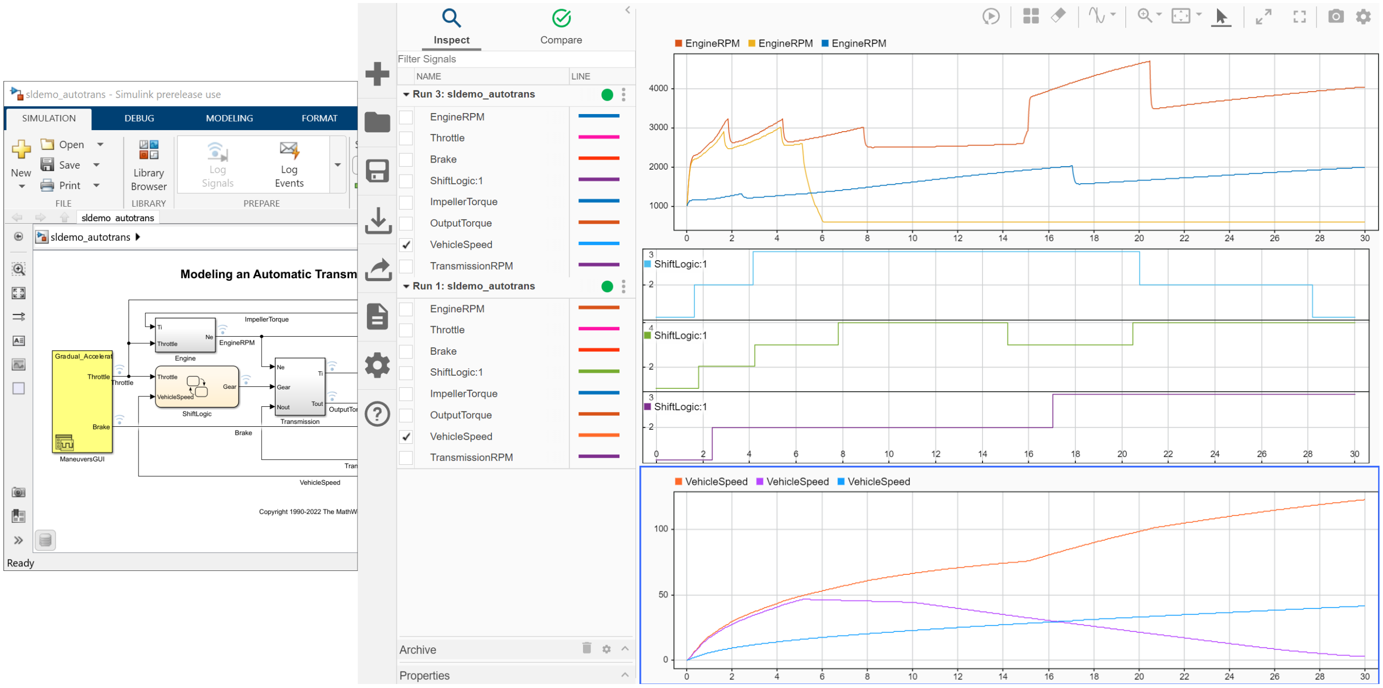
   Run a simulation in a model configured to log data to the Simulation Data Inspector, or import data from the workspace or a MAT-file. You can view and verify model input data or inspect logged simulation data while iteratively modifying your model diagram, parameter values, or model configuration.
2. "Inspect Simulation Data"

   Plot signals on multiple subplots, zoom in and out on specified plot axes, and use data cursors to understand and evaluate the data. "Create Plots Using the Simulation Data Inspector" to tell your story.
3. "Compare Simulation Data"

   Compare individual signals or simulation runs and analyze your comparison results with relative, absolute, and time tolerances. The compare tools in the Simulation Data Inspector facilitate iterative design and allow you to highlight signals that do not meet your tolerance requirements. For more information about the comparison operation, see "How the Simulation Data Inspector Compares Data".
4. "Save and Share Simulation Data Inspector Data and Views"

   Share your findings with others by saving Simulation Data Inspector data and views.

You can also harness the capabilities of the Simulation Data Inspector from the command line. For more information, see "Inspect and Compare Data Programmatically".



Open the Simulation Data Inspector

- Simulink Toolstrip: On the **Simulation** tab, under **Review Results**, click **Data Inspector**.
- Click the streaming badge on a signal to open the Simulation Data Inspector and plot the signal.
- MATLAB command prompt: Enter `Simulink.sdi.view`.

Examples

Apply a Tolerance to a Signal in Multiple Runs

You can use the Simulation Data Inspector programmatic interface to modify a parameter for the same signal in multiple runs. This example adds an absolute tolerance of `0.1` to a signal in all four runs of data.

First, clear the workspace and load the Simulation Data Inspector session with the data. The session includes logged data from four simulations of a Simulink® model of a longitudinal controller for an aircraft.

```
Simulink.sdi.clear
Simulink.sdi.load('AircraftExample.mldatx');
```

Use the `Simulink.sdi.getRunCount` function to get the number of runs in the Simulation Data Inspector. You can use this number as the index for a for loop that operates on each run.

```
count = Simulink.sdi.getRunCount;
```

Then, use a for loop to assign the absolute tolerance of `0.1` to the first signal in each run.

```
for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    aircraftRun = Simulink.sdi.getRun(runID);
    sig = getSignalByIndex(aircraftRun,1);
    sig.AbsTol = 0.1;
end
```

- “View Data in the Simulation Data Inspector”
- “Inspect Simulation Data”
- “Compare Simulation Data”
- “Iterate Model Design Using the Simulation Data Inspector”

Programmatic Use

`Simulink.sdi.view` opens the Simulation Data Inspector from the MATLAB command line.

Version History

Introduced in R2010b

See Also

Functions

`Simulink.sdi.clear` | `Simulink.sdi.clearPreferences` | `Simulink.sdi.snapshot`

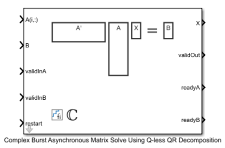
Topics

“View Data in the Simulation Data Inspector”
“Inspect Simulation Data”
“Compare Simulation Data”
“Iterate Model Design Using the Simulation Data Inspector”

Blocks

Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition

Compute the value of X in the equation $A'AX = B$ for complex-valued matrices using asynchronous Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition block solves the system of linear equations $A'AX = B$ using asynchronous Q-less QR decomposition, where A and B are complex-valued matrices.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A' A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

This block operates asynchronously. The forward- and backward-substitution and Q-less QR decomposition run independently using the latest R and B matrices.

Ports

Input

A(i,:) — Rows of complex matrix A
vector

Rows of complex matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B — Rows of complex matrix B
vector

Rows of complex matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether `A(i,:)` input is valid
`Boolean` scalar

Whether `A(i,:)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (`true`), the block captures the values at the `A(i,:)` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

validInB — Whether `B` input is valid
`Boolean` scalar

Whether `B` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `B` input port is valid. When this value is `1` (`true`) and the `readyB` value is `1` (`true`), the block captures the values at the `B` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `X` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

readyA — Whether block is ready for input `A(i,:)`
`Boolean` scalar

Whether block is ready for input `A(i,:)`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validInA` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input `B`
`Boolean` scalar

Whether block is ready for input `B`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validInB` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
4 (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix B — Number of columns in matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

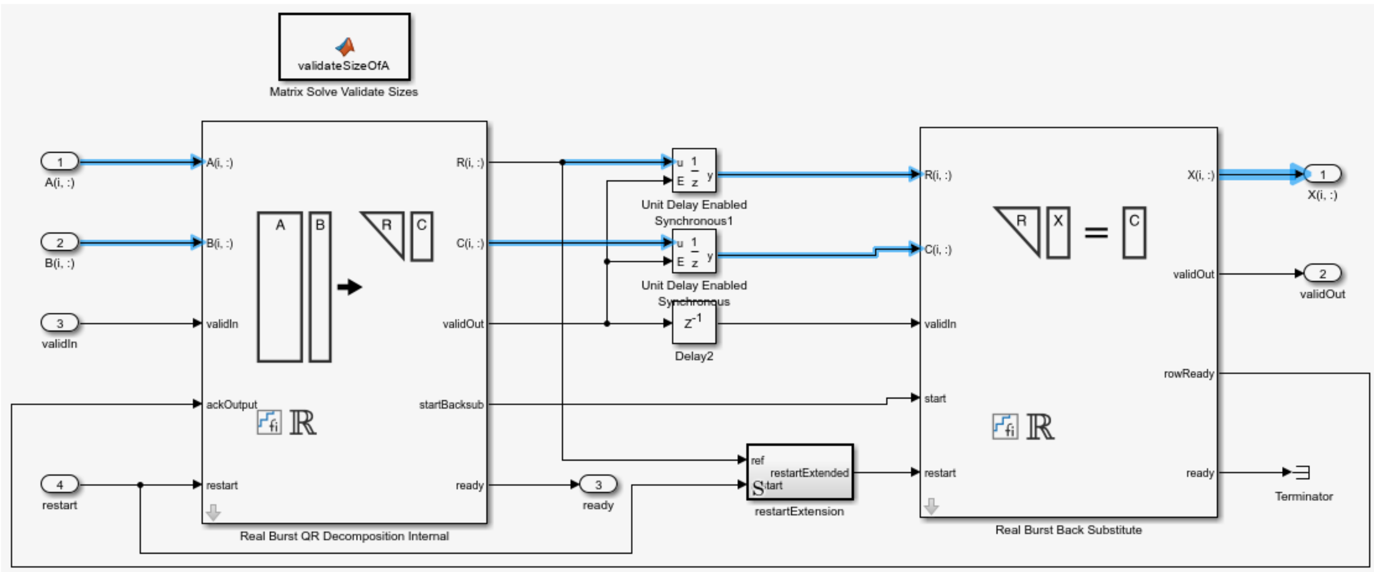
Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

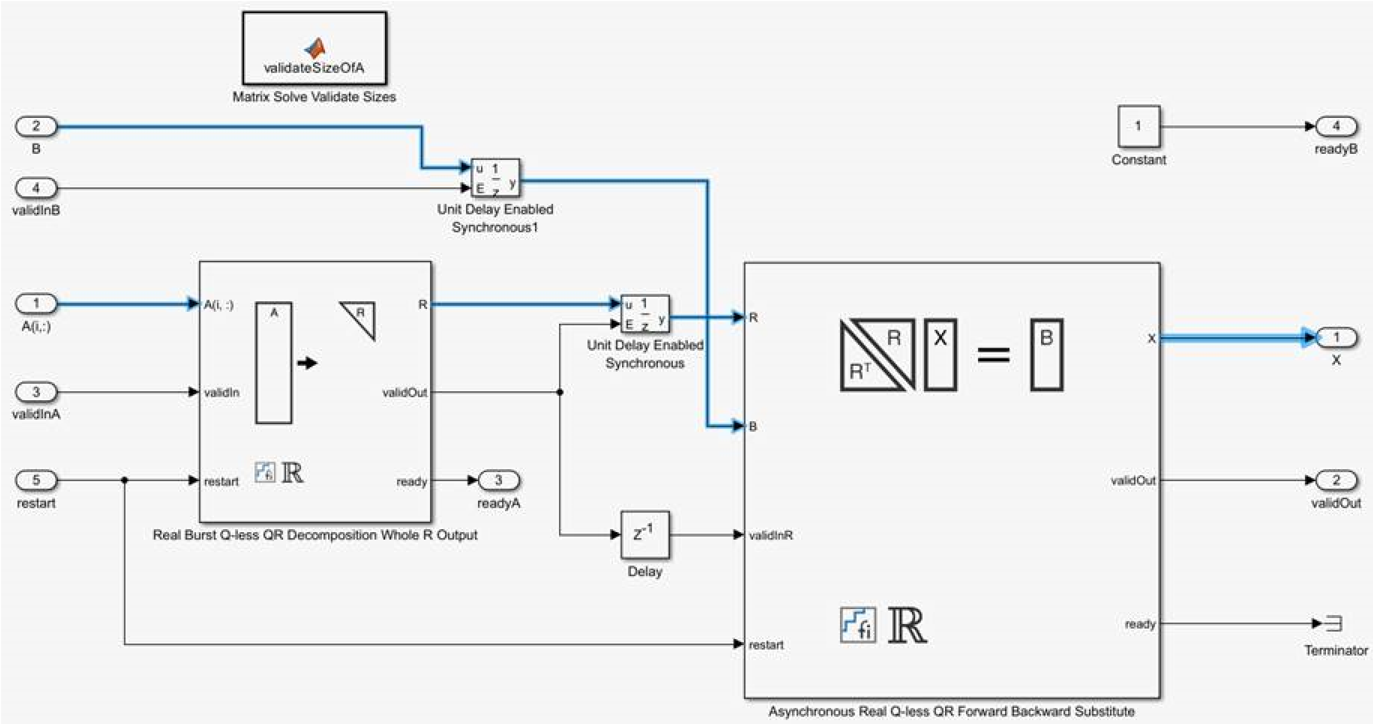
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input A matrix and the resulting R matrix is put into a buffer. Then, a forward backward substitution block uses the input B matrix and the buffered R matrix to compute $R'RX = B$. Because the R and B matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first R and B matrices are available. Then it runs continuously using the latest buffered R and B matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing A and B matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of R and B matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

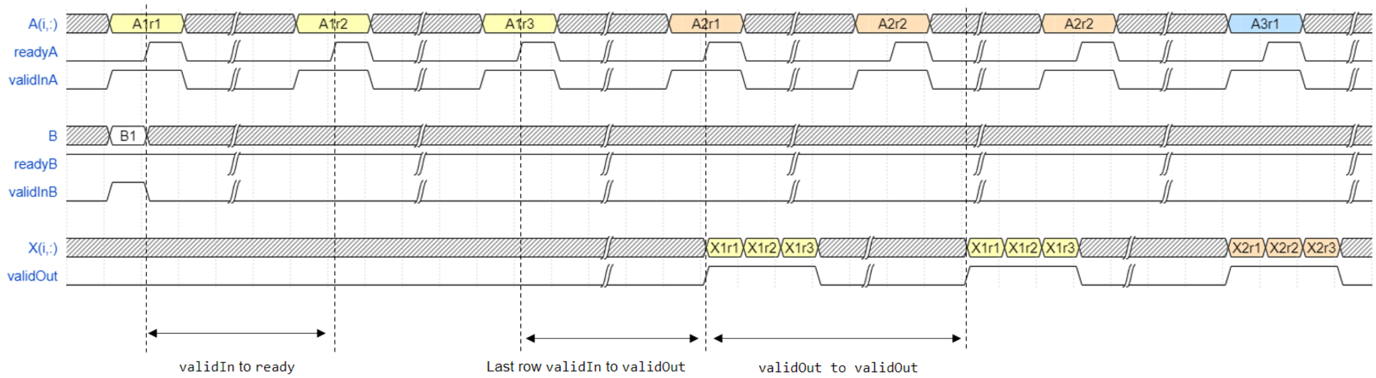
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Asynchronous Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

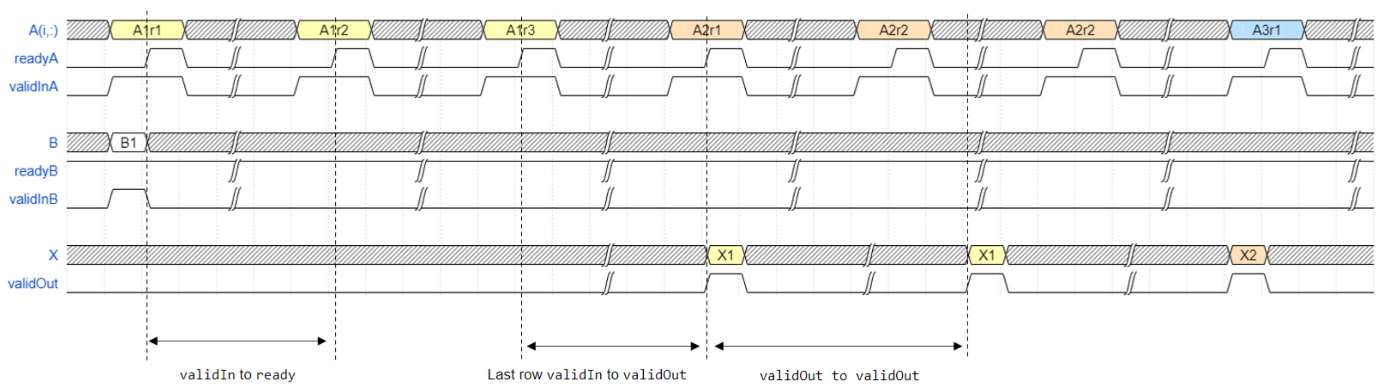


In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.

- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	`validOut` to `validOut` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl+5)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl*2 + 11)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx® Zynq® UltraScale™ + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado® v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:

  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	36070	425280	8.48
CLB Registers	45878	850560	5.39
DSPs	12	4272	0.28
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.781 ns
Slack	0.199 ns
Clock Frequency	263.09 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

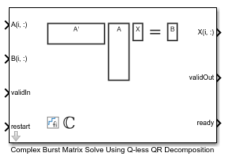
See Also

Blocks
Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition | Complex Burst Matrix Solve Using QR Decomposition | Complex Burst Matrix Solve Using Q-less QR Decomposition

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Complex Burst Matrix Solve Using Q-less QR Decomposition

Compute the value of X in the equation $A'AX = B$ for complex-valued matrices using Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Burst Matrix Solve Using Q-less QR Decomposition block solves the system of linear equations, $A'AX = B$, using Q-less QR decomposition, where A and B are complex-valued matrices.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst Matrix Solve Using Q-less QR Decomposition block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of matrix *B*
vector

Rows of matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is `1` (`true`) and the `ready` value is `1` (`true`), the block captures the values at the `A(i,:)` and `B(i,:)` input ports. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X(i,:) — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X(i,:)` is valid. When this value is `1` (`true`), the block has successfully computed a row of *X*. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and the `validIn` value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
`4` (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
`0` (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector

Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Tips

Use `fixed.getQlessQRMatrixSolveModel(A,B)` to generate a template model containing a Complex Burst Matrix Solve Using Q-less QR Decomposition block for complex-valued input matrices `A` and `B`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

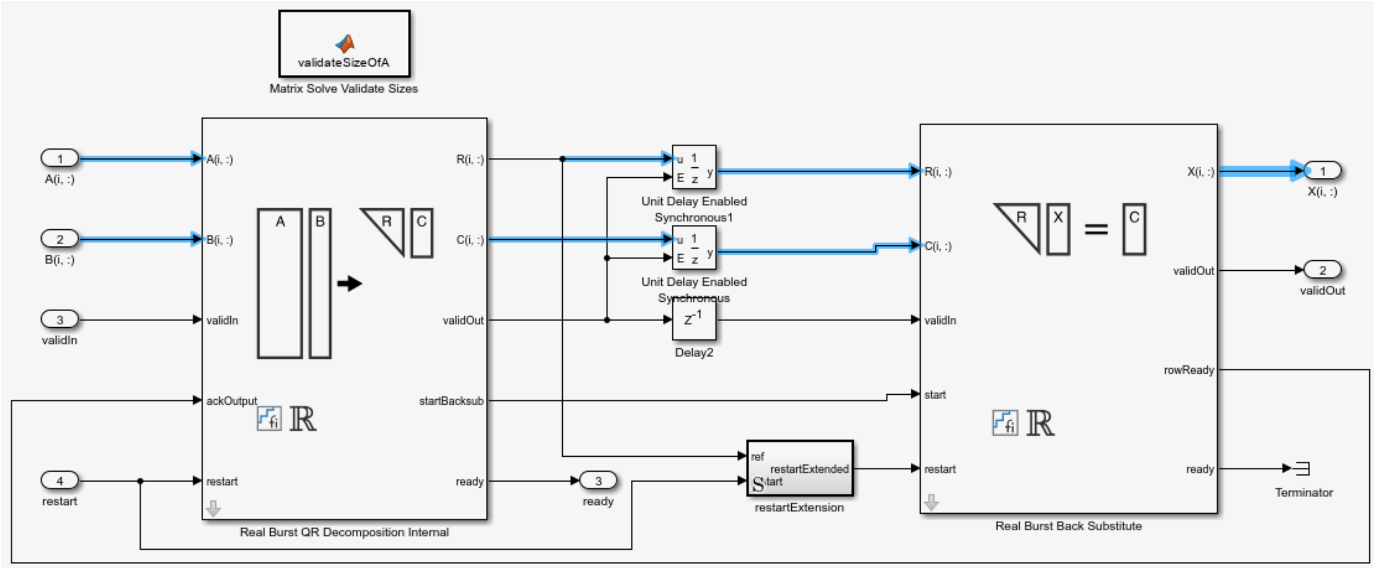
Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

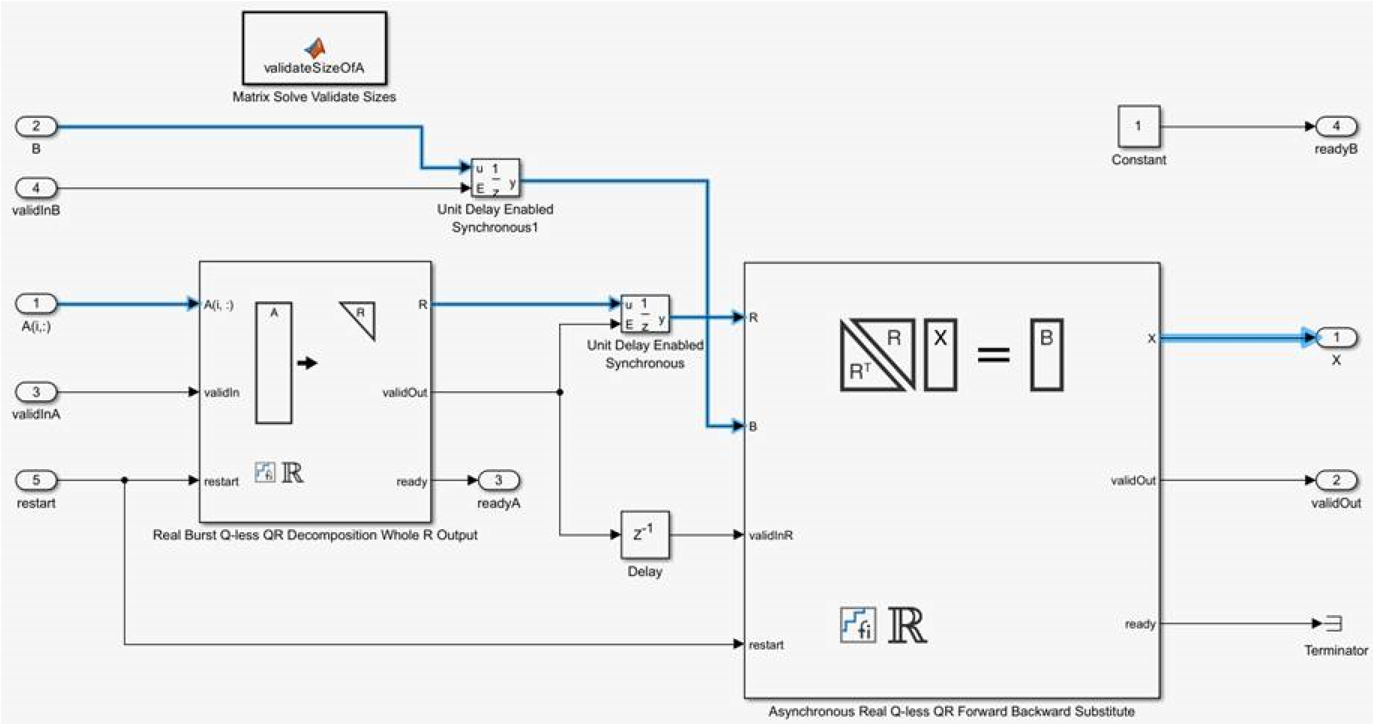
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input *A* matrix and the resulting *R* matrix is put into a buffer. Then, a forward backward substitution block uses the input *B* matrix and the buffered *R* matrix to compute $R'RX = B$. Because the *R* and *B* matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first *R* and *B* matrices are available. Then it runs continuously using the latest buffered *R* and *B* matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing *A* and *B* matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of *R* and *B* matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

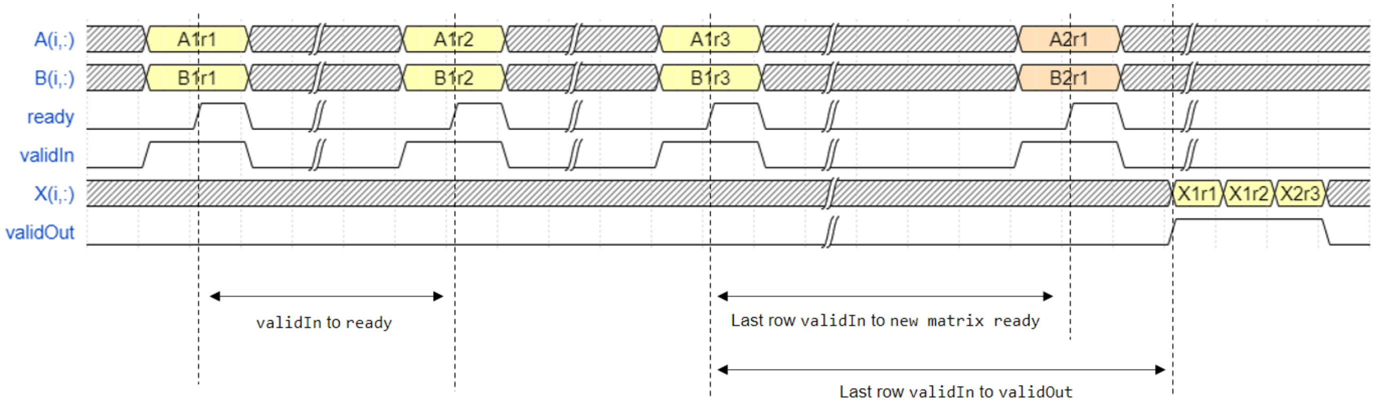
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

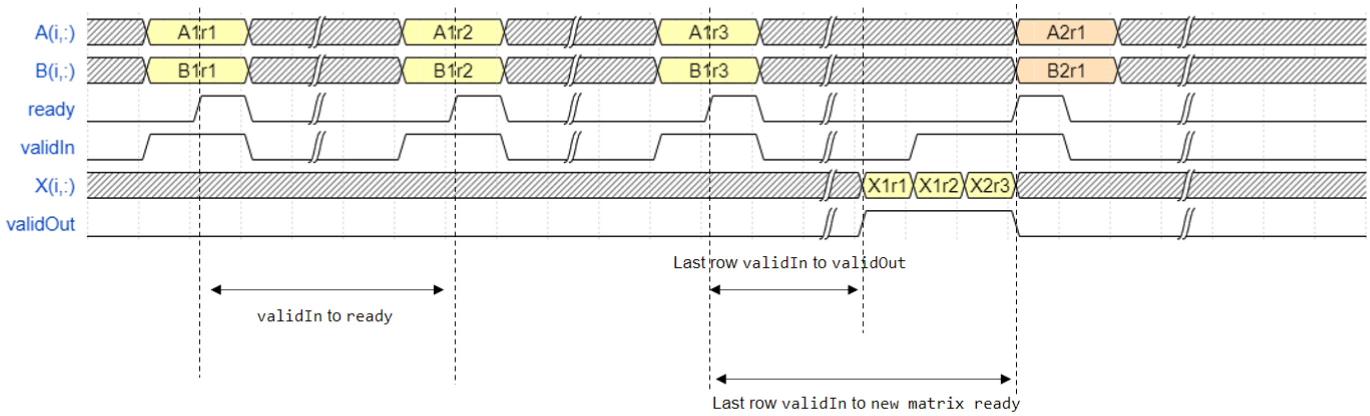


In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Matrix Solve Using Q-less QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)	Last row `validIn` to new matrix ready (cycles)
Real Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$(wl + 5)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl + 5)*n + 3.5*(n - 1)^2 + (n - 1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Complex Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$(wl*2 + 11)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl*2 + 11)*n + 3.5*(n-1)^2 + (n-1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Real Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$
Complex Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$7*2^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`

    - Matrix *A* dimension: 16-by-16
    - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	30915	425280	7.27
CLB Registers	34833	850560	4.10
DSPs	12	4272	0.28
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.686 ns
Slack	0.296 ns
Clock Frequency	269.98 MHz

Version History

Introduced in R2020a

R2022a: Support for Tikhonov regularization parameter

The Complex Burst Matrix Solve Using Q-less QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0 .

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

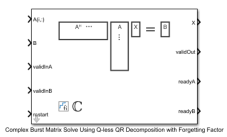
Blocks
Real Burst Matrix Solve Using Q-less QR Decomposition | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Complex Burst Matrix Solve Using QR Decomposition

Functions
`fixed.qlessQRMatrixSolve`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Compute the value of X in the equation $A'AX = B$ for complex-valued matrices with infinite number of rows using asynchronous Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block solves the system of linear equations $A'AX = B$ using asynchronous Q-less QR decomposition, where A and B are complex-valued matrices. A is an infinitely tall matrix representing streaming data.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

This block operates asynchronously. The forward- and backward-substitution and Q-less QR decomposition run independently using the latest R and B matrices.

Ports

Input

A(i,:) — Rows of complex matrix A
vector

Rows of complex matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B — Rows of complex matrix B
vector

Rows of complex matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether `A(i,:)` input is valid
`Boolean` scalar

Whether `A(i,:)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (`true`), the block captures the values at the `A(i,:)` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

validInB — Whether `B` input is valid
`Boolean` scalar

Whether `B` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `B` input port is valid. When this value is `1` (`true`) and the `readyB` value is `1` (`true`), the block captures the values at the `B` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `X` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

readyA — Whether block is ready for input A(i,:)
Boolean scalar

Whether block is ready for input A(i,:), returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (true) and validInA is 1 (true), the block accepts input data in the next time step. When this value is 0 (false), the block ignores input data in the next time step.

After sending a true validInA signal, there may be some delay before readyA is set to false. To ensure all data is processed, you must wait until readyA is set to false before sending another true validInA signal.

Data Types: Boolean

readyB — Whether block is ready for input B
Boolean scalar

Whether block is ready for input B, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (true) and validInB is 1 (true), the block accepts input data in the next time step. When this value is 0 (false), the block ignores input data in the next time step.

After sending a true validInB signal, there may be some delay before readyB is set to false. To ensure all data is processed, you must wait until readyB is set to false before sending another true validInB signal.

Data Types: Boolean

Parameters

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
4 (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: n
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix B — Number of columns in matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: p
Type: character vector
Values: positive integer-valued scalar
Default: 1

Forgetting factor — Forgetting factor applied after each row of the matrix is factored
0.99 (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of *A* is input indefinitely.

Programmatic Use
Block Parameter: `forgetting_factor`
Type: character vector
Values: real positive scalar
Default: `0`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

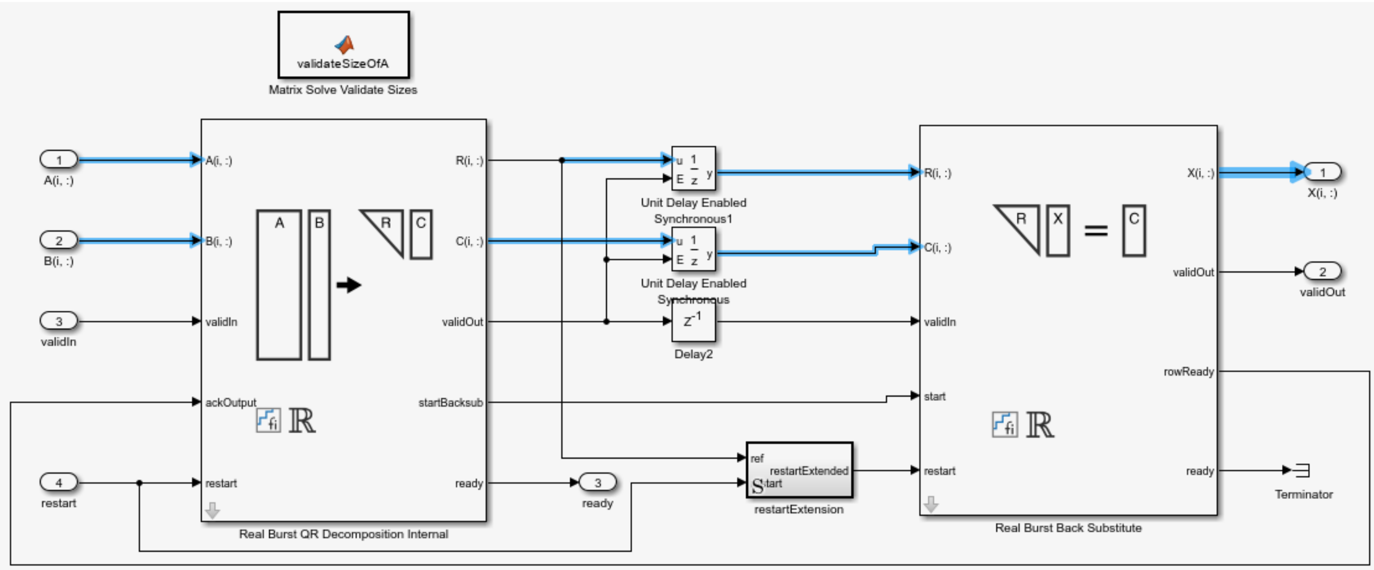
Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

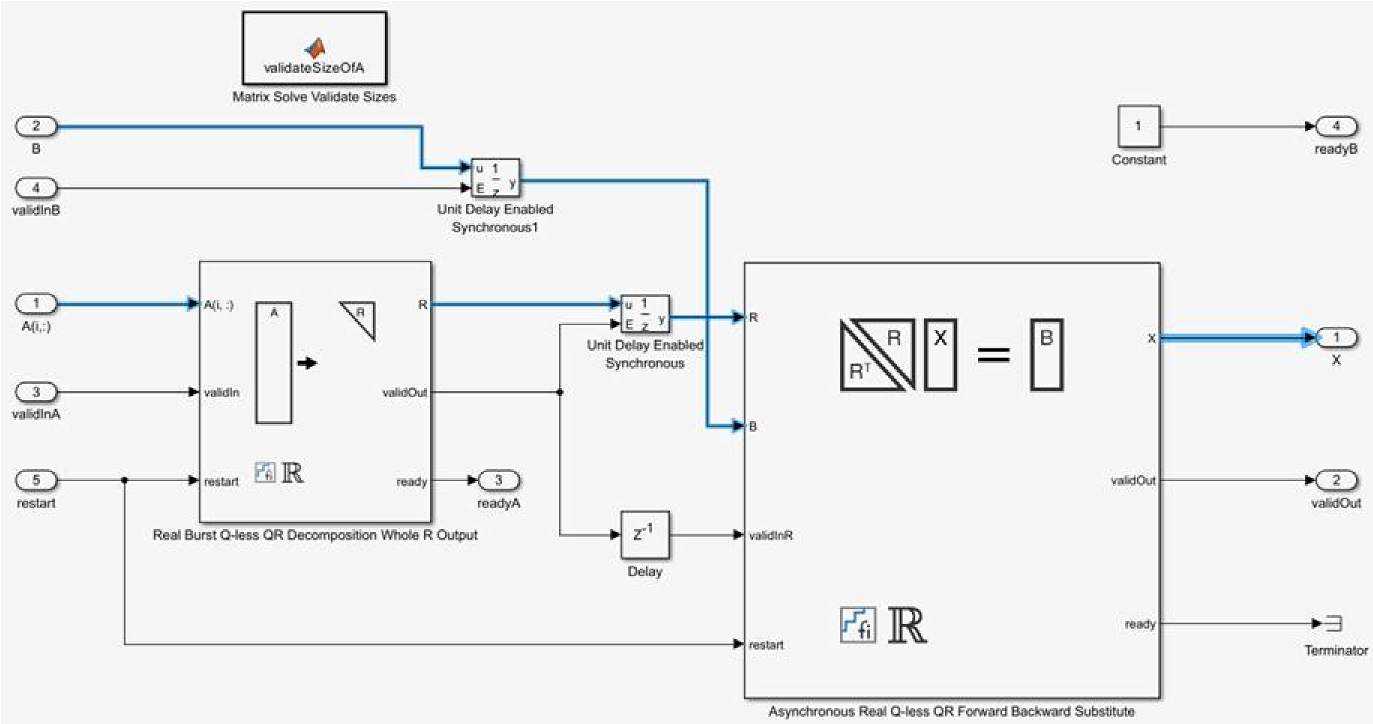
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input A matrix and the resulting R matrix is put into a buffer. Then, a forward backward substitution block uses the input B matrix and the buffered R matrix to compute $R'RX = B$. Because the R and B matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first R and B matrices are available. Then it runs continuously using the latest buffered R and B matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing A and B matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of R and B matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

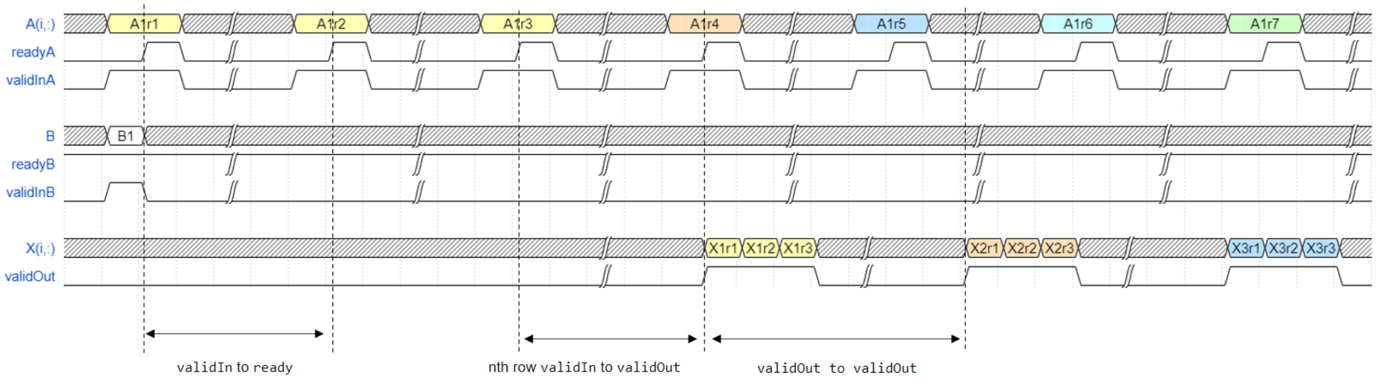
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

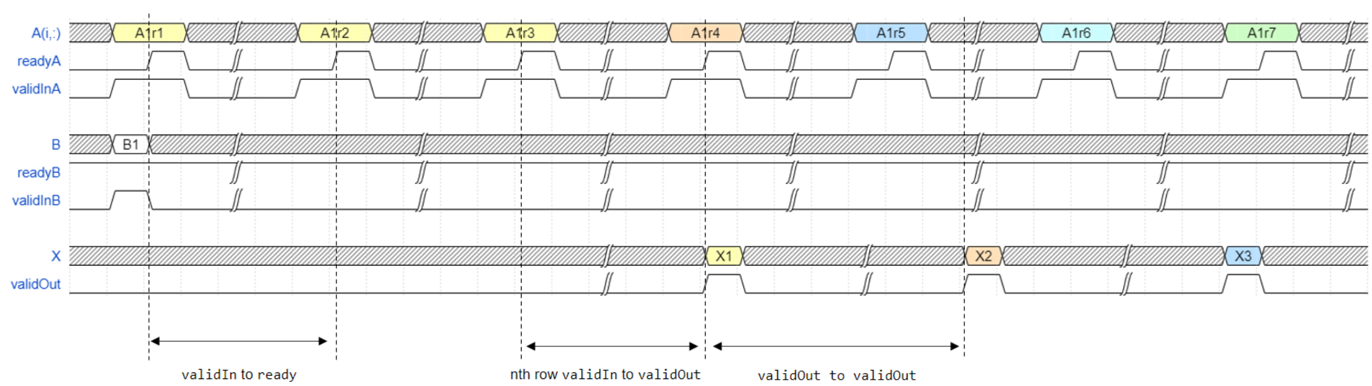


In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- n^{th} row `validIn` to `validOut` — From the n^{th} row input to the block starting to output the first solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.

- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* and *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks.

Block	**Operation**	**`validIn` to `ready` (cycles)**	**`validOut` to `validOut` (cycles)**	**n^{th} Row `validIn` to `validOut` (cycles)**
Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl + 5)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl*2 + 11)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n+ 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: inf-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	36045	425280	8.48
CLB Registers	45870	850560	5.39
DSPs	44	4272	1.03
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.794 ns
Slack	0.187 ns
Clock Frequency	262.26 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

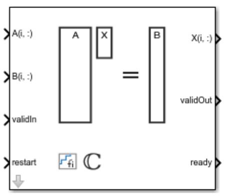
See Also

Blocks
Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Complex Burst Matrix Solve Using Q-less QR Decomposition

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Complex Burst Matrix Solve Using QR Decomposition

Compute the value of x in the equation $Ax = B$ for complex-valued matrices using QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Burst Matrix Solve Using QR Decomposition block solves the system of linear equations $Ax = B$ using QR decomposition, where A and B are complex-valued matrices. To compute $x = A^{-1}$, set B to be the identity matrix.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst Matrix Solve Using QR Decomposition block computes the matrix solution of complex-valued $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ where λ is the regularization parameter, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of matrix *B*
vector

Rows of matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values at the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X(i,:) — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed a row of matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in matrices *A* and *B*
4 (default) | positive integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in matrix *A*
`4` (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: positive integer-valued scalar
Default: `0`

Output datatype — Data type of the output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`

Default: `'fixdt(1,18,14)'`

Tips

Use `fixed.getMatrixSolveModel(A,B)` to generate a template model containing a Complex Burst Matrix Solve Using QR Decomposition block for complex-valued input matrices `A` and `B`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

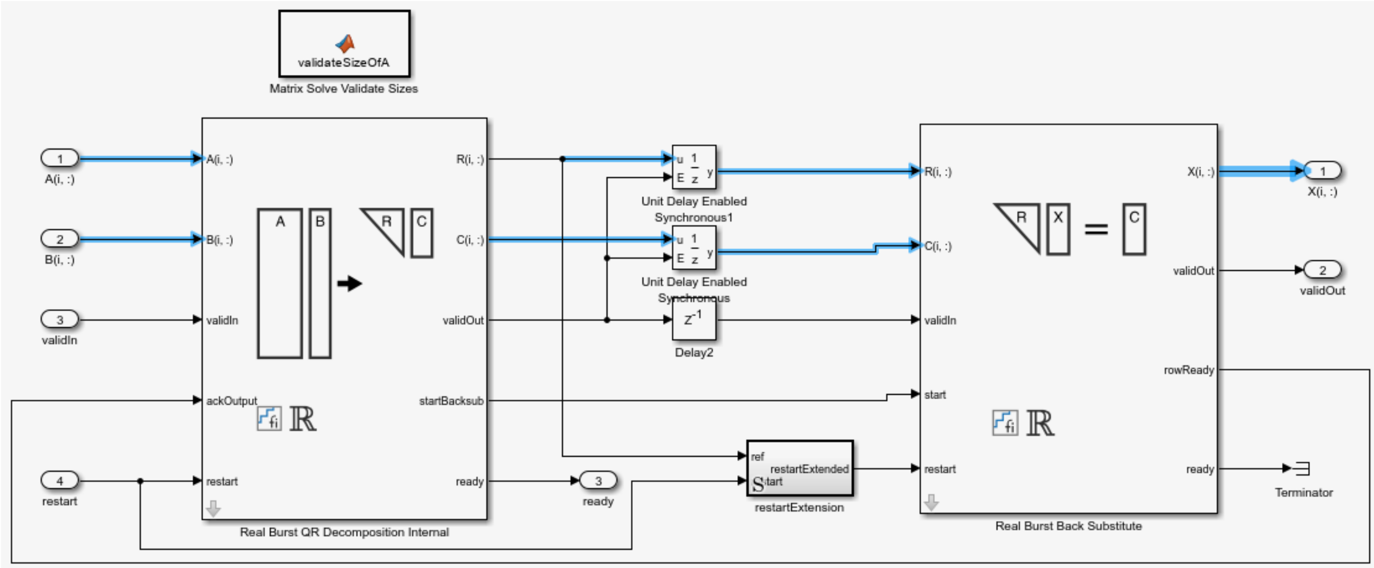
Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

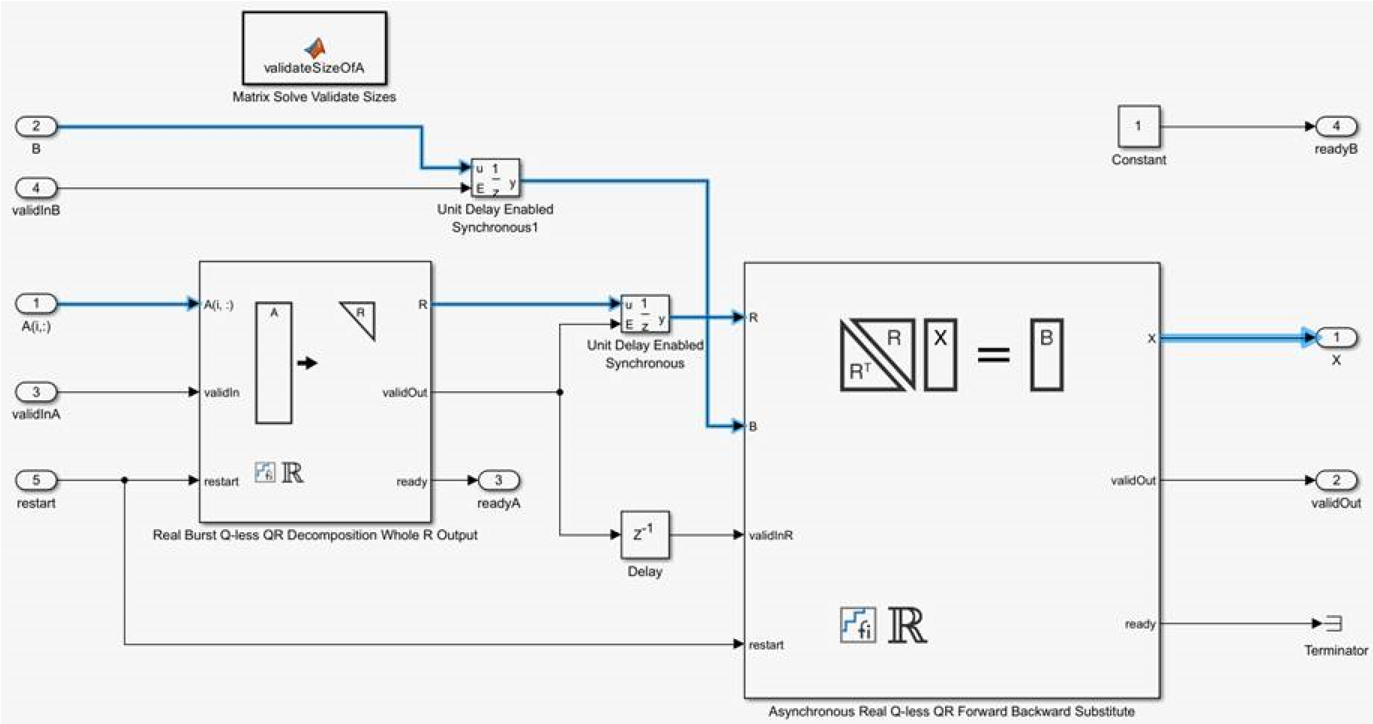
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input A matrix and the resulting R matrix is put into a buffer. Then, a forward backward substitution block uses the input B matrix and the buffered R matrix to compute $R'RX = B$. Because the R and B matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first R and B matrices are available. Then it runs continuously using the latest buffered R and B matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing A and B matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of R and B matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

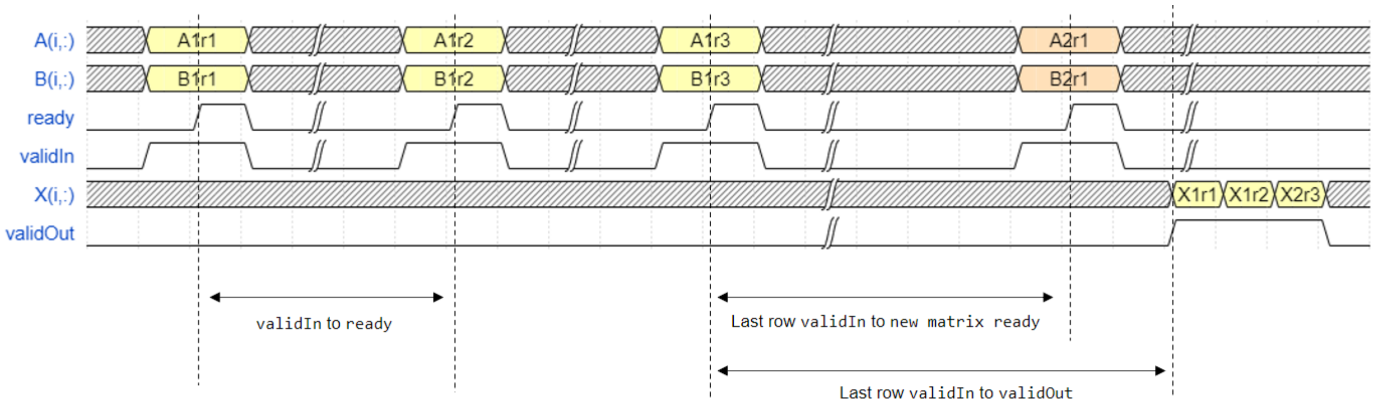
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

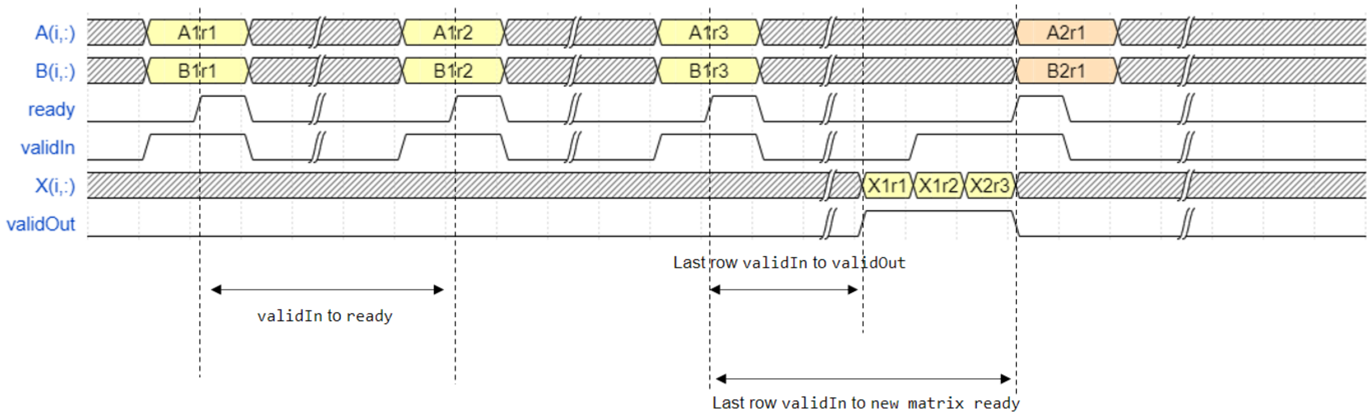


In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Matrix Solve Using Q-less QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	**Operation**	**`validIn` to ready (cycles)**	**Last Row `validIn` to `validOut` (cycles)**	**Last row `validIn` to new matrix ready (cycles)**
Real Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$(wl + 5)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl + 5)*n + 3.5*(n - 1)^2 + (n - 1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Complex Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$(wl*2 + 11)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl*2 + 11)*n + 3.5*(n-1)^2 + (n-1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Real Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$
Complex Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$7*2^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`

- Matrix *A* dimension: 16-by-16
- Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	23719	425280	5.58
CLB Registers	24062	850560	2.83
DSPs	6	4272	0.14
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.242 ns
Slack	0.072 ns
Clock Frequency	306.62 MHz

Version History

Introduced in R2019b

R2022a: Support for Tikhonov regularization parameter

The Complex Burst Matrix Solve Using QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0 .

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

Blocks

Real Burst Matrix Solve Using QR Decomposition | Real Burst Matrix Solve Using Q-less QR Decomposition | Complex Partial-Systolic Matrix Solve Using QR Decomposition

Functions

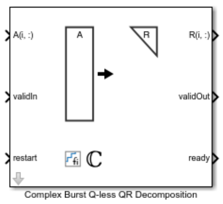
`fixed.qrMatrixSolve`

Topics

"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Burst Q-less QR Decomposition

Q-less QR decomposition for complex-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Burst Q-less QR Decomposition block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, where A is a complex-valued matrix, without computing Q. The solution to $A'Ax = B$ is $x = R\backslash R'\backslash b$.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst Q-less QR Decomposition block computes the upper-triangular factor R of the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the regularization parameter.

Ports

Input

A(i,:) — Rows of complex matrix *A*
vector

Rows of complex matrix `A`, specified as a vector. A is a m-by-n matrix where $m \geq 2$ and $n \geq 2$. If A is a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data at the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R(i,:) — Rows of upper-triangular matrix *R*
scalar | vector

Rows of the economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper-triangular matrix. The size of the matrix *R* is min(m,n)-by-n. The output at `R(i,:)` has the same data type as the input at `A(i,:)`.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and the `validIn` value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A — Number of columns in matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: n
Type: character vector
Values: positive integer-valued scalar
Default: 4

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: regularizationParameter
Type: character vector
Values: real nonnegative scalar
Default: 0

Tips

Use `fixed.getQlessQRDecompositionModel(A)` to generate a template model containing a Complex Burst Q-less QR Decomposition block for complex-valued input matrix A.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

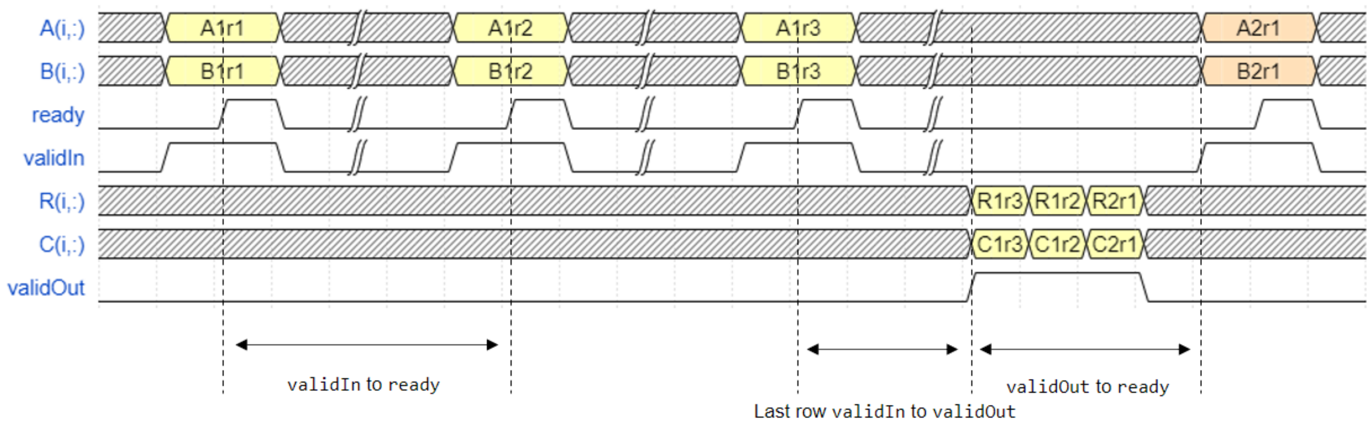
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *R* and *C* matrices row by row continuously. The matrices are output from the last row to the first row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

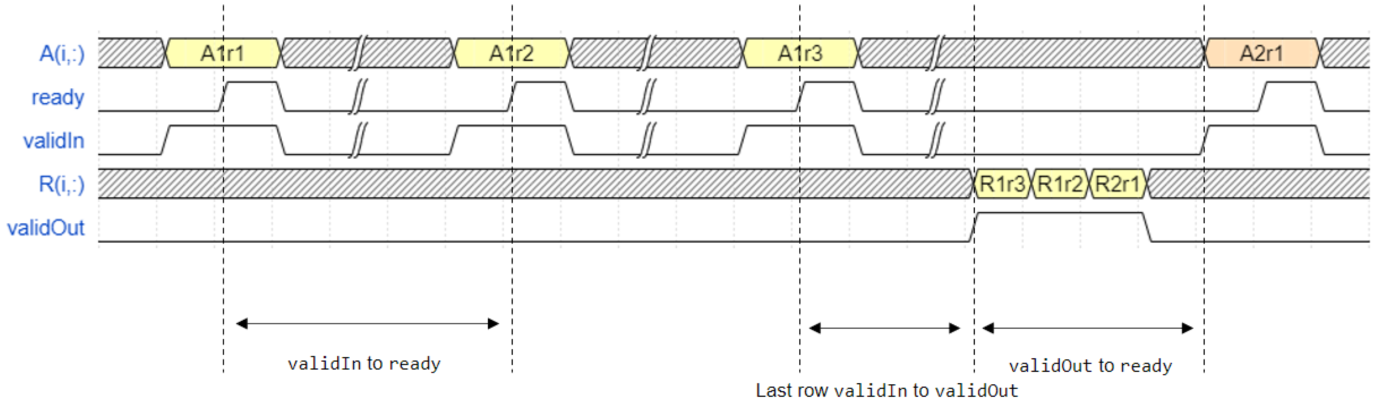


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the matrix *R* row by row continuously. The matrix is output from the last row to the first row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst QR Decomposition* blocks.

Block	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)	`validOut` to `ready` (cycles)
Real Burst QR Decomposition	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2$	$\min(m,n) + 1$
Complex Burst QR Decomposition	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2$	$\min(m,n) + 1$
Real Burst Q-less QR Decomposition	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2$	$\min(m,n) + 1$
Complex Burst Q-less QR Decomposition	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2$	$\min(m,n) + 1$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - Matrix *A* dimension: 16-by-16
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	21137	425280	4.97
CLB Registers	21157	850560	2.49
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.18 ns
Slack	0.134 ns
Clock Frequency	312.57 MHz

Version History

Introduced in R2020a

R2022a: Support for Tikhonov regularization parameter

The Complex Burst Q-less QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0.

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

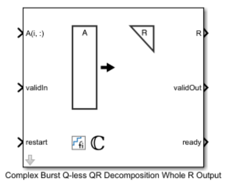
Blocks
Real Burst Q-less QR Decomposition | Complex Partial-Systolic Q-less QR Decomposition | Complex Burst QR Decomposition

Functions
`fixed.qlessQR`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Burst Q-less QR Decomposition Whole R Output

Q-less QR decomposition for complex-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Burst Q-less QR Decomposition Whole R Output block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, where A is a complex-valued matrix, without computing Q. The solution to $A'Ax = B$ is $x = R\backslash R'\backslash b$.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst Q-less QR Decomposition Whole R Output block computes the upper-triangular factor R of the economy size QR decomposition of $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ where λ is the regularization parameter.

Ports

Input

A(i,:) — Rows of complex matrix *A*
vector

Rows of complex matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If A is a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Economy size QR decomposition matrix *R*
vector

Economy size QR decomposition matrix *R*, returned as a vector. *R* is an upper triangular matrix. The size of matrix *R* is `min(m,n)`-by-`n`. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in input matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Regularization parameter — Regularization parameter
`0` (default) | real nonnegative scalar

Regularization parameter, specified as a real nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

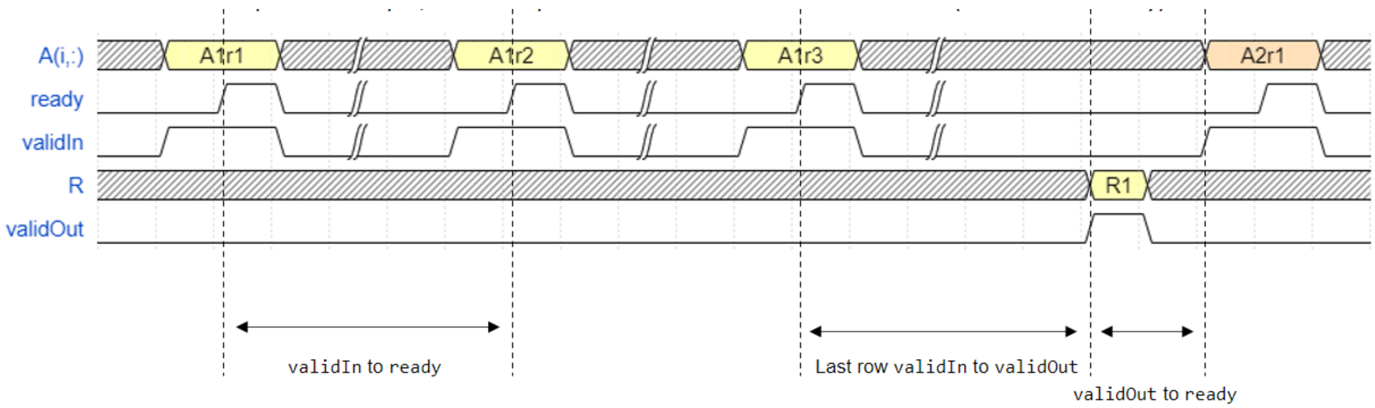
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Q-less QR Decomposition Whole R Output* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *R* matrix as a single vector.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Q-less QR Decomposition Whole R Output* blocks.

Block	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)	`validOut` to `ready` (cycles)
Real Burst Q-less QR Decomposition Whole R Output	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2 + \min(m,n) - 1$	2
Complex Burst Q-less QR Decomposition Whole R Output	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2 + \min(m,n) -1$	2

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - Matrix *A* dimension: 16-by-16
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	21177	425280	4.98
CLB Registers	21153	850560	2.49
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.172 ns
Slack	0.142 ns
Clock Frequency	313.35 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

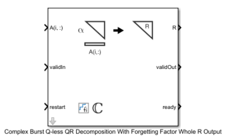
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Real Burst Q-less QR Decomposition Whole R Output | Complex Burst Q-less QR Decomposition | Complex Burst QR Decomposition

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output

Q-less QR decomposition for complex-valued matrices with infinite number of rows



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition, $A = QR$, without computing Q. A is an infinitely tall complex-valued matrix representing streaming data.

When the regularization parameter is nonzero, the Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output block initializes the first upper-triangular factor R to λI_n before factoring in the rows of A, where λ is the regularization parameter and I_n = eye(n)

Ports

Input

A(i,:) — Rows of complex matrix A
vector

Rows of complex matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If A uses a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Economy size QR decomposition matrix *R*
vector

Economy size QR decomposition matrix *R*, returned as a vector. *R* is an upper triangular matrix. The size of matrix *R* is n-by-n. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of columns in matrix A — Number of columns in input matrix *A*
`4` (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: n
Type: character vector
Values: positive integer-valued scalar
Default: 4

Forgetting factor — Forgetting factor applied after each row of the matrix is factored
`0.99` (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of *A* is input indefinitely.

Programmatic Use
Block Parameter: `forgetting_factor`
Type: character vector
Values: real positive scalar
Default: `0`

Regularization parameter — Regularization parameter
`0` (default) | real nonnegative scalar

Regularization parameter, specified as a real nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

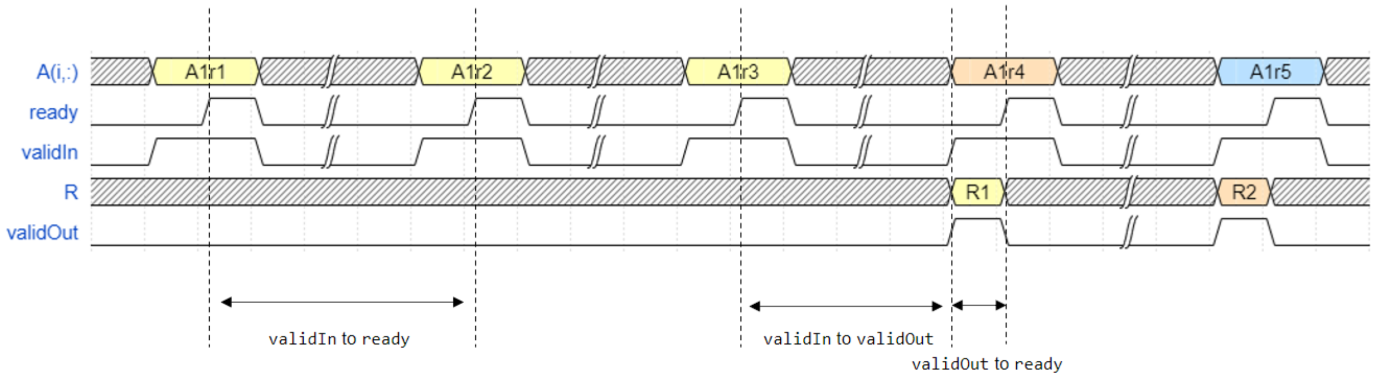
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Q-less QR Decomposition with Forgetting Factor Whole R Output* blocks accept and process the matrix *A* row by row. After accepting the first *m* rows, the block starts to output the *R* matrix as a vector. Then, for each row input, the block calculates an *R* matrix.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- `validIn` to `validOut` — From a successful row input to the block starting to output the corresponding solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Q-less QR Decomposition with Forgetting Factor Whole R Output* blocks.

Block	`validIn` to `ready` (cycles)	`validIn` to `validOut` (cycles)	`validOut` to `ready` (cycles)
Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output	(*wl* + 5)*n + 2 + *n*	(*wl* + 5)*n + 2 + *n* - 1	1
Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output	(*wl**2 + 11)*n + 2 + *n*	(*wl**2 + 11)*n + 2 + *n* - 1	1

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `n = 16`
  - Matrix *A* dimension: inf-by-16
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	21142	425280	4.97
CLB Registers	21158	850560	2.49
DSPs	32	4272	0.75
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.051 ns
Slack	0.264 ns
Clock Frequency	325.80 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

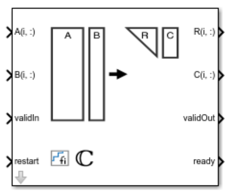
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output | Complex Burst Q-less QR Decomposition Whole R Output

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Complex Burst QR Decomposition

QR decomposition for complex-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Burst QR Decomposition block uses QR decomposition to compute R and $C = Q'B$, where $QR = A$, and A and B are complex-valued matrices. The least-squares solution to $Ax = B$ is $x = R\backslash C$. R is an upper triangular matrix and Q is an orthogonal matrix. To compute $C = Q'$, set B to be the identity matrix.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Burst QR Decomposition block transforms $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ in-place to $R = Q'\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ and $\begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$ in-place to $C = Q'\begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$ where λ is the regularization parameter, QR is the economy size QR decomposition of $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of matrix *B*
vector

Rows of matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values on the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`), and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R(i,:) — Rows of matrix *R*
scalar | vector

Rows of the economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper triangular matrix. The size of the matrix *R* is min(*m*,*n*)-by-*n*. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

C(i,:) — Rows of matrix *C=Q'B*
scalar | vector

Rows of the economy size QR decomposition matrix *C=Q'B*, returned as a scalar or vector. *C* has the same number of rows as *R*. *C* has the same data type as *B*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at output ports `R(i,:)` and `C(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the *R* and *C* matrices. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`), and the `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in matrices A and B

4 (default) | positive integer-valued scalar

The number of rows in matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in matrix A
4 (default) | positive integer-valued scalar

The number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix B
1 (default) | positive integer-valued scalar

The number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Tips

Use `fixed.getQRDecompositionModel(A,B)` to generate a template model containing a Complex Burst QR Decomposition block for complex-valued input matrices `A` and `B`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

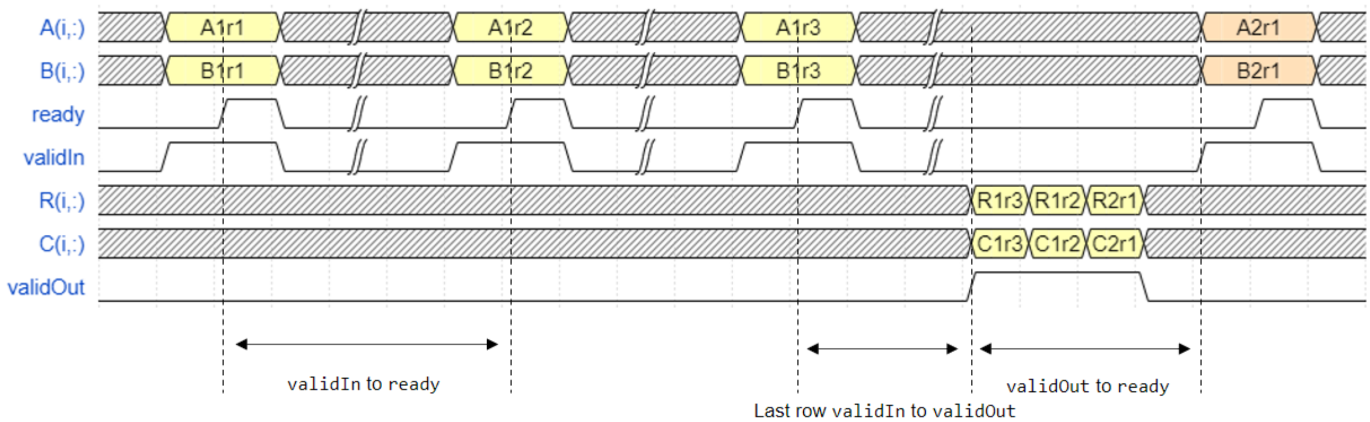
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst QR Decomposition* blocks accept and process A and B matrices row by row synchronously. After accepting m rows, the block outputs the R and C matrices row by row continuously. The matrices are output from the last row to the first row.

For example, assume that the input A and B matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

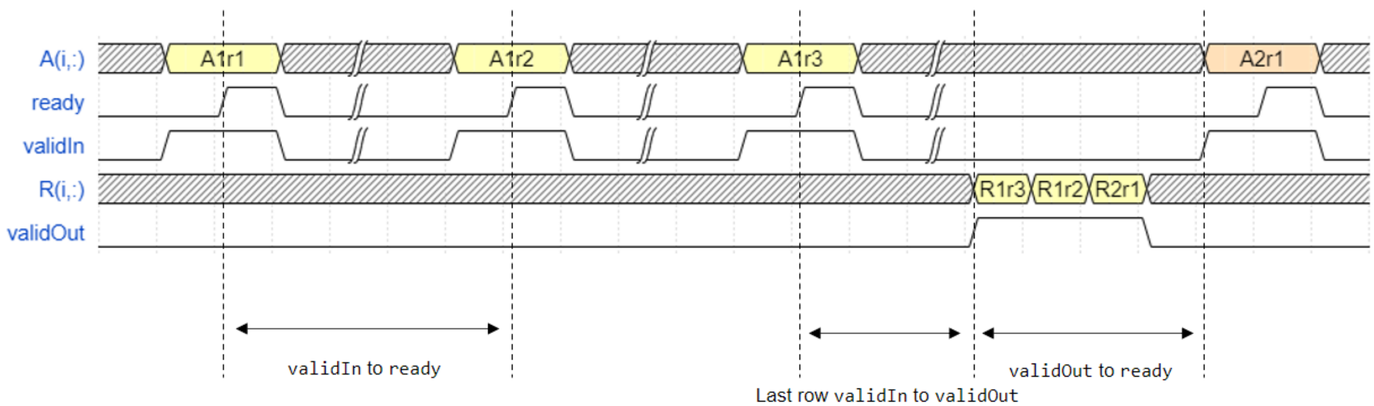


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the matrix *R* row by row continuously. The matrix is output from the last row to the first row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst QR Decomposition* blocks.

Block	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)	`validOut` to `ready` (cycles)
Real Burst QR Decomposition	(*wl* + 5)*min(*m*,*n*) + 2	(*wl* + 5)*min(*m*,*n*) + 2	min(*m*,*n*) + 1
Complex Burst QR Decomposition	(*wl**2 + 11)*min(*m*,*n*) + 2	(*wl**2 + 11)*min(*m*,*n*) + 2	min(*m*,*n*) + 1
Real Burst Q-less QR Decomposition	(*wl* + 5)*min(*m*,*n*) + 2	(*wl* + 5)*min(*m*,*n*) + 2	min(*m*,*n*) + 1
Complex Burst Q-less QR Decomposition	(*wl**2 + 11)*min(*m*,*n*) + 2	(*wl**2 + 11)*min(*m*,*n*) + 2	min(*m*,*n*) + 1

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	22713	425280	5.34

Resource	Usage	Available	Utilization (%)
CLB Registers	22469	850560	2.64
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.149 ns
Slack	0.166 ns
Clock Frequency	315.72 MHz

Version History

Introduced in R2019b

R2022a: Support for Tikhonov regularization parameter

The Complex Burst QR Decomposition block now supports the Tikhonov "Regularization parameter" on page 2-0.

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

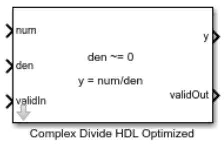
Blocks
Real Burst QR Decomposition | Complex Burst Q-less QR Decomposition | Complex Partial-Systolic QR Decomposition

Functions
`fixed.qrAB`

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Complex Divide HDL Optimized

Divide one input by another and generate optimized HDL code



Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Complex Divide HDL Optimized block outputs the result of dividing the scalar **num** by the scalar **den**, such that **y** = **num**/**den**.

Limitations

Data type override is not supported for the Complex Divide HDL Optimized block.

Ports

Input

num — Numerator
scalar

Numerator, specified as a scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

den — Denominator
scalar

Denominator, specified as a scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

validIn — Whether input is valid
`Boolean` scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the **num** and **den** input ports are valid. When this value is `1` (`true`), the block captures the values at the input ports **num** and **den**. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

y — Output computed by dividing inputs
complex scalar

Output computed by dividing **num** by **den**, such that **y** = **num**/**den**, returned as a complex scalar with data type specified by **Output datatype**.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output at port **y**. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Parameters

Output datatype — Data type of output
`fixdt(1,18,10)` (default) | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of output **y**, specified as `fixdt(1,18,10)`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly or expressed as a data type object, such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,10)'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,10)'`

Tips

The blocks Divide by Constant HDL Optimized, Real Divide HDL Optimized, and Complex Divide HDL Optimized all perform the division operation and generate optimized HDL code.

- Real Divide HDL Optimized and Complex Divide HDL Optimized are based on a CORIDC algorithm. These blocks accept a wide variety of inputs, but will result in greater latency.
- Divide by Constant HDL Optimized accepts only real inputs and a constant divisor. Use of this block consumes DSP slices, but will complete the division operation in fewer cycles and at a higher clock rate.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers.

Fully Pipelined Fixed-Point Computations

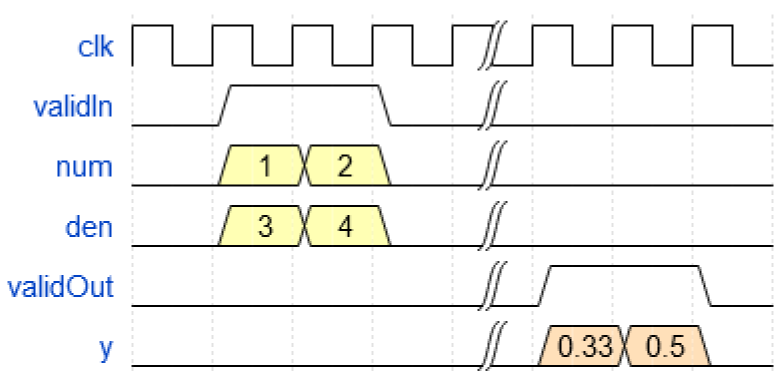
The Complex Divide HDL Optimized block supports HDL code generation for fixed-point data with binary-point scaling. It is designed with this application in mind, and employs hardware specific semantics and optimizations. One of these optimizations is pipelining its entire internal circuitry to maintain a very high throughput.

When deploying intricate algorithms to FPGA or ASIC devices, there is often a trade-off between resource usage and total throughput for a given computation. Resource-sharing often reduces the resources consumed by a design, but also reduces the throughput in the process. Simple arithmetic and trigonometric computations, which typically form parts of bigger computations, require high throughput to drive circuits further in the design. Thus, fully pipelined implementations consume more on-chip resources but are beneficial in large designs.

All of the key computational units in the Complex Divide HDL Optimized block are fully pipelined internally. This includes not only the CORDIC circuitry used to perform the Givens rotations, but also the adders and shifters used elsewhere in the design, thus ensuring maximum throughput.

How to Interface with the Complex Divide HDL Optimized Block

Because of its fully pipelined nature, the Complex Divide HDL Optimized block is able to accept input data on any cycle, including consecutive cycles. To send input data to the block, the **validIn** signal must be set to true. When the block has finished the computation and is ready to send the output, it will set **validOut** to true for one clock cycle. For inputs sent on consecutive cycles, **validOut** will also be set to true on consecutive cycles. Both the numerator and the denominator must be sent together on the same cycle.



Division by Zero Behavior

For fixed-point inputs **num** and **den**, the Complex Divide HDL Optimized block saturates on overflow for division by zero. The behavior for fixed-point division by zero is summarized in the table below.

Wrap Overflow	Saturate Overflow
0/0 = 0	0/0 = 0
1/0 = 0	1/0 = upper bound
-1/0 = 0	-1/0 = lower bound

For floating-point inputs, the Complex Divide HDL Optimized block follows IEEE® Standard 754.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Restrictions

Supports binary-point scaled fixed-point data types only.

See Also

Blocks

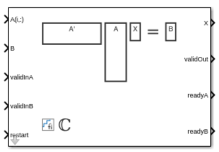
Real Divide HDL Optimized | Real Reciprocal HDL Optimized | Normalized Reciprocal HDL Optimized

Functions

`fixed.cordicReciprocal` | `fixed.cordicDivide`

Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition

Compute the value of X in the equation $A'AX = B$ for complex-valued matrices using Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition block solves the system of linear equations, $A'AX = B$, using Q-less QR decomposition, where A and B are complex-valued matrices.

When "Regularization parameter" on page 2-0 is nonzero, the Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of matrix A
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

B — Matrix B
vector

Matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

Complex Number Support: Yes

validInA — Whether input A is valid
`Boolean` scalar

Whether input `A` is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value at `readyA` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

validInB — Whether input B is valid
`Boolean` scalar

Whether input `B` is valid, specified as a Boolean scalar. This control signal indicates when the data from the `B` input port is valid. When this value is 1 (`true`) and the value at `readyB` is 1 (`true`), the block captures the values at the `B` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X — Matrix *X*
matrix | vector

Matrix *X*, returned as a vector or matrix.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X` is valid. When this value is 1 (`true`), the block has successfully computed a row of matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

readyA — Whether block is ready for input A
`Boolean` scalar

Whether the block is ready for input A, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validInA` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input B
`Boolean` scalar

Whether the block is ready for input B, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validInB` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
4 (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix B — Number of columns in matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in matrix B, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix X
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix X, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

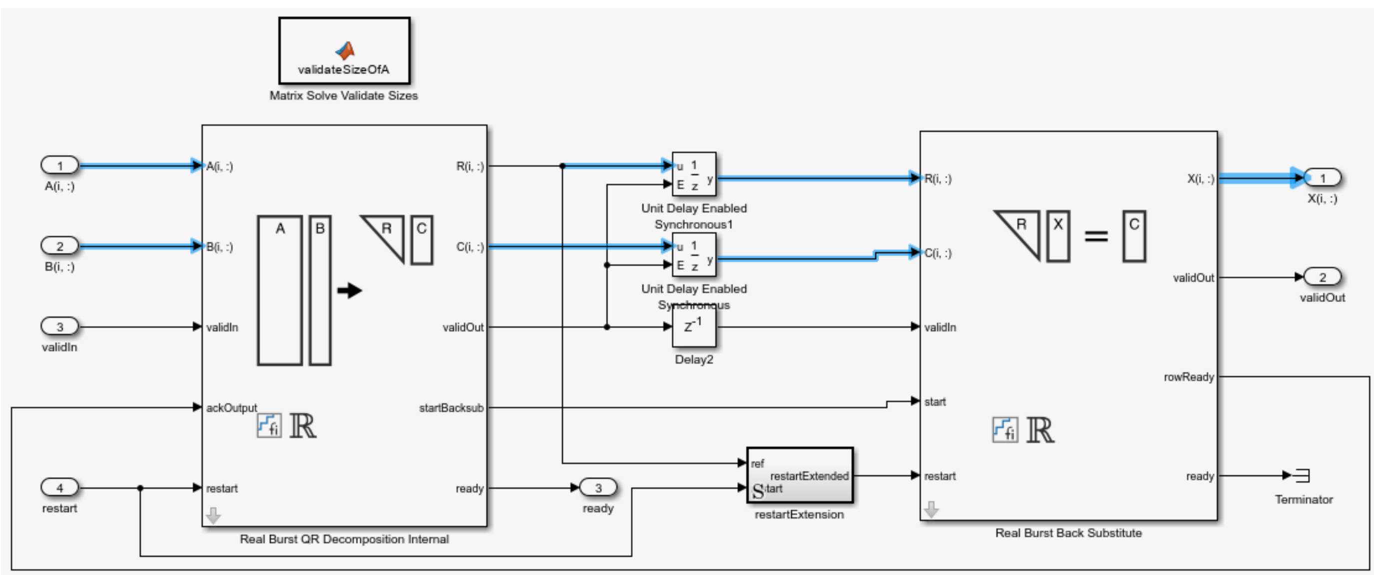
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

AMBA AXI Handshake Process

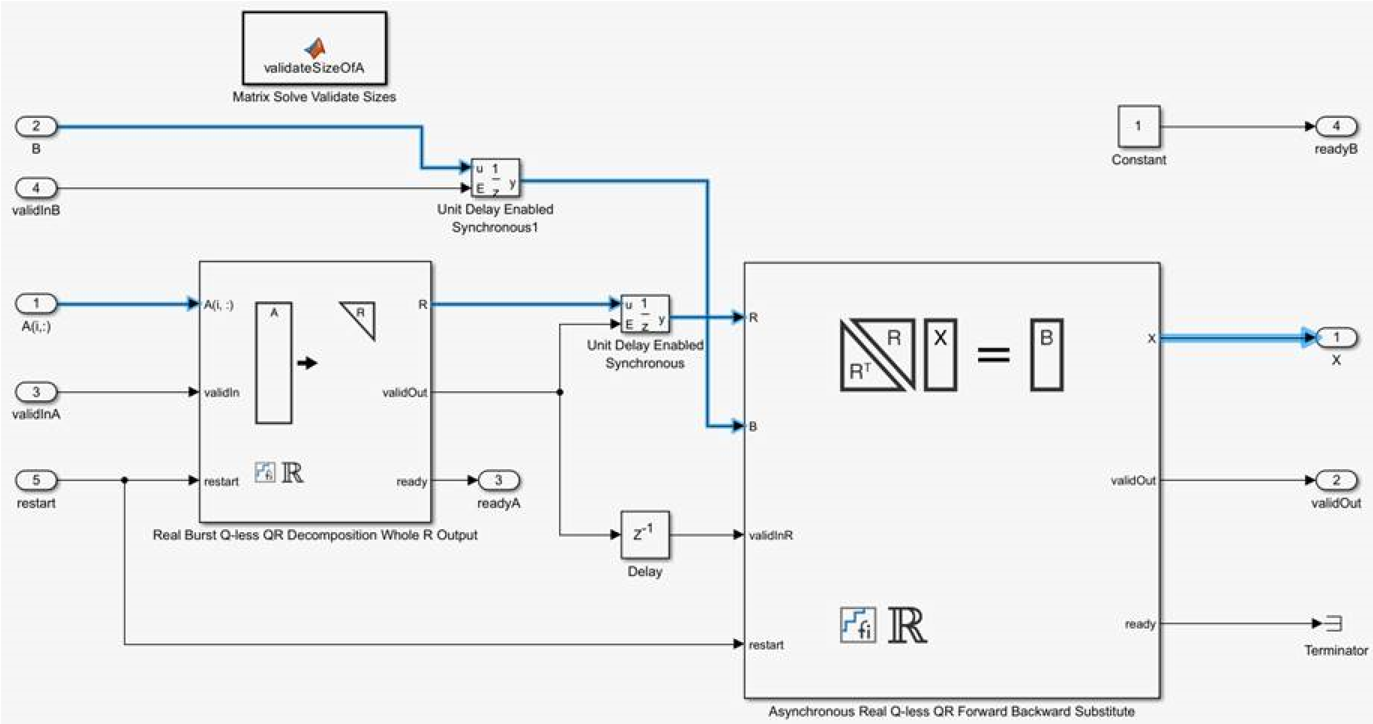
This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input A matrix and the resulting R matrix is put into a buffer. Then, a forward backward substitution block uses the input B matrix and the buffered R matrix to compute $R'RX = B$. Because the R and B matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first R and B matrices are available. Then it runs continuously using the latest buffered R and B matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing A and B matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of R and B matrices.

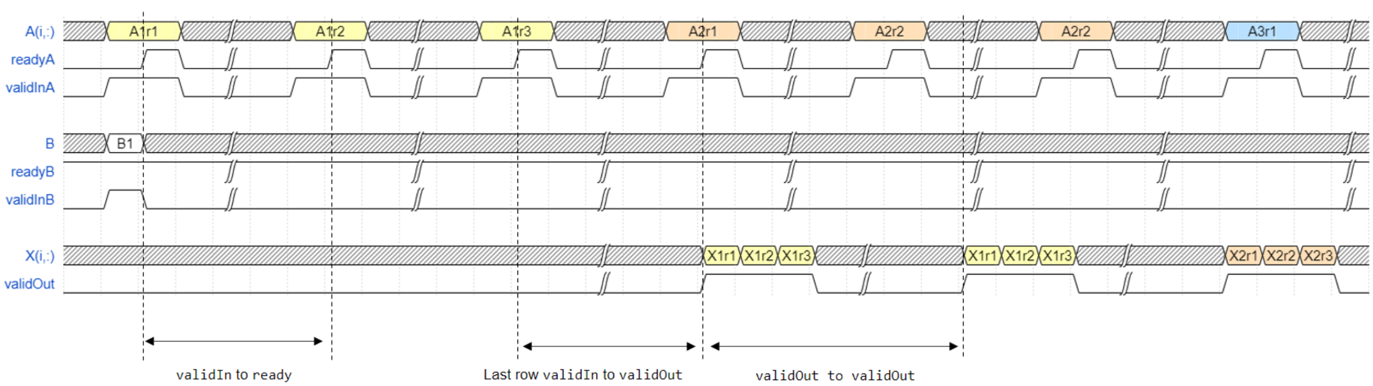


The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

Block Timing

The *Burst Asynchronous Matrix Solve Using Q-less QR Decomposition* blocks accept matrix A row-by-row and matrix B as a single vector. After accepting the first valid pair of A and B matrices, the block outputs the X matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input A matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



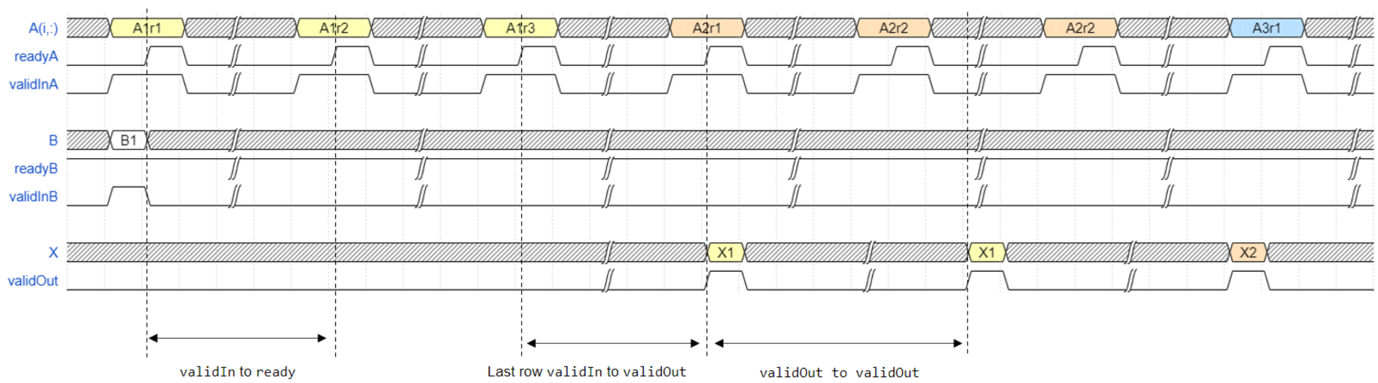
In the figure,

- `A1r1` is the first row of the first A matrix, `A1r2` is the second row of the first A matrix, and so on.

- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	`validOut` to `validOut` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl+5)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl*2 + 11)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`

    - Matrix *A* dimension: 16-by-16
    - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	300675	425280	70.70
CLB Registers	260811	850560	30.66
DSPs	12	4272	0.28
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.954 ns
Slack	0.029 ns
Clock Frequency	251.83 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

This block depends on a partial-systolic QR decomposition block. Since 23a, when you update the diagram, the loop which composes the partial-systolic pipeline in the QR decomposition block is unrolled. This updated internal architecture removes dead operations in simulation and generated code, thus requiring fewer hardware resources. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

R2022a: Support for Tikhonov regularization parameter

The Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0 .

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

Blocks
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Complex Burst Matrix Solve Using Q-less QR Decomposition

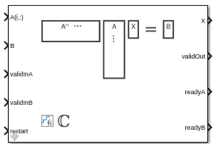
Functions
`fixed.qlessQRMatrixSolve`

Topics

"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Compute the value of X in the equation $A'AX = B$ for complex-valued matrices with infinite number of rows using Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block solves the system of linear equations, $A'AX = B$, using Q-less QR decomposition, where A and B are complex-valued matrices. A is an infinitely tall matrix representing streaming data.

When the regularization parameter is nonzero, the Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor initializes the first upper-triangular factor R to λI_n before factoring in the rows of A, where λ is the regularization parameter and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

B — Matrix *B*
matrix | vector

Matrix B, specified as a vector or a matrix. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether A input is valid
`Boolean` scalar

Whether `A(i, :)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (true), the block captures the values at the A(i,:) input port. When this value is 0 (false), the block ignores the input samples.

After sending a true validInA signal, there may be some delay before readyA is set to false. To ensure all data is processed, you must wait until readyA is set to false before sending another true validInA signal.

Data Types: Boolean

validInB — Whether input B is valid
Boolean scalar

Whether input B is valid, specified as a Boolean scalar. This control signal indicates when the data from the B input port is valid. When this value is 1 (true) and the readyB value is 1 (true), the block captures the values at the B input port. When this value is 0 (false), the block ignores the input samples.

After sending a true validInB signal, there may be some delay before readyB is set to false. To ensure all data is processed, you must wait until readyB is set to false before sending another true validInB signal.

Data Types: Boolean

restart — Whether to clear internal states
Boolean scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (true), the block stops the current calculation and clears all internal states. When this value is 0 (false) and the validInA and validInB values are 1 (true), the block begins a new subframe.

Data Types: Boolean

Output

X — Matrix *X*
matrix | vector

Matrix *X*, returned as a matrix or vector.

Data Types: single | double | fixed point

validOut — Whether output data is valid
Boolean scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port X is valid. When this value is 1 (true), the block has successfully computed a row of *X*. When this value is 0 (false), the output data is not valid.

Data Types: Boolean

readyA — Whether block is ready for input A
Boolean scalar

Whether the block is ready for input A, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (true) and validInA value is 1 (true), the block accepts input data in the next time step. When this value is 0 (false), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input B
`Boolean` scalar

Whether the block is ready for input B, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validInB` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
`4` (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Forgetting factor — Forgetting factor applied after each row of the matrix is factored
0.99 (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of *A* is input indefinitely.

Programmatic Use
Block Parameter: `forgettingFactor`
Type: character vector
Values: positive integer-valued scalar

Default: `0.99`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Tips

- Use `fixed.forgettingFactor` to compute the forgetting factor, α, for an infinite number of rows with the equivalent gain of a matrix with m rows.
- Use `fixed.forgettingFactorInverse` to compute the number of rows, m, of a matrix with equivalent gain corresponding to forgetting factor α

  .

Algorithms

Q-less QR Decomposition with Forgetting Factor

The Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block implements the following recursion to compute the upper-triangular factor R of continuously streaming n-by-1 row vectors $A(k,:)$ using forgetting factor α. It's as if matrix A is infinitely tall. The forgetting factor in the range $0 < \alpha < 1$ prevents it from integrating without bound.

$$
\begin{aligned}
R_0 &= \text{zeros}(n, n) \\
[\sim, R_1] &= \text{qr}\left(\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix}, 0\right) \\
R_1 &= \alpha R_1 \\
[\sim, R_2] &= \text{qr}\left(\begin{bmatrix} R_1 \\ A(2,:) \end{bmatrix}, 0\right) \\
R_2 &= \alpha R_2 \\
&\vdots \\
[\sim, R_k] &= \text{qr}\left(\left[\begin{bmatrix} R_{k-1} \\ A(k,:) \end{bmatrix}\right], 0\right) \\
R_k &= \alpha R_k \\
&\vdots
\end{aligned}
$$

Q-less QR Decomposition with Forgetting Factor and Tikhonov Regularization

The output X_k after processing the k^{th} input $A(k,:)$ is computed using the following iteration.

$$
\begin{aligned}
R_0 &= \lambda I_n \\
[\sim, R_1] &= \text{qr}\left(\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix}, 0\right) \\
R_1 &= \alpha R_1 \\
X_1 &= R_1 \backslash (R'_1 \backslash B) \\
[\sim, R_2] &= \text{qr}\left(\begin{bmatrix} R_1 \\ A(2,:) \end{bmatrix}, 0\right) \\
R_2 &= \alpha R_2 \\
X_2 &= R_2 \backslash (R'_2 \backslash B) \\
&\vdots \\
[\sim, R_k] &= \text{qr}\left(\begin{bmatrix} R_{k-1} \\ A(k,:) \end{bmatrix}, 0\right) \\
R_k &= \alpha R_k \\
X_k &= R_k \backslash (R'_k \backslash B) \\
&\vdots
\end{aligned}
$$

This is mathematically equivalent to computing $A'_k A_k X = B$, where A_k is defined as follows, though the block never actually creates A_k.

$$
A_k = \begin{bmatrix} \alpha^k \lambda I_n \\ \begin{bmatrix} \alpha^k & & & \\ & \alpha^{k-1} & & \\ & & \ddots & \\ & & & \alpha \end{bmatrix} A(1:k,:) \end{bmatrix}
$$

Forward and Backward Substitution

When an upper triangular factor is ready, then forward and backward substitution are computed with the current input B to produce output X.

$$X = R_k\backslash(R_k'\backslash B)$$

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

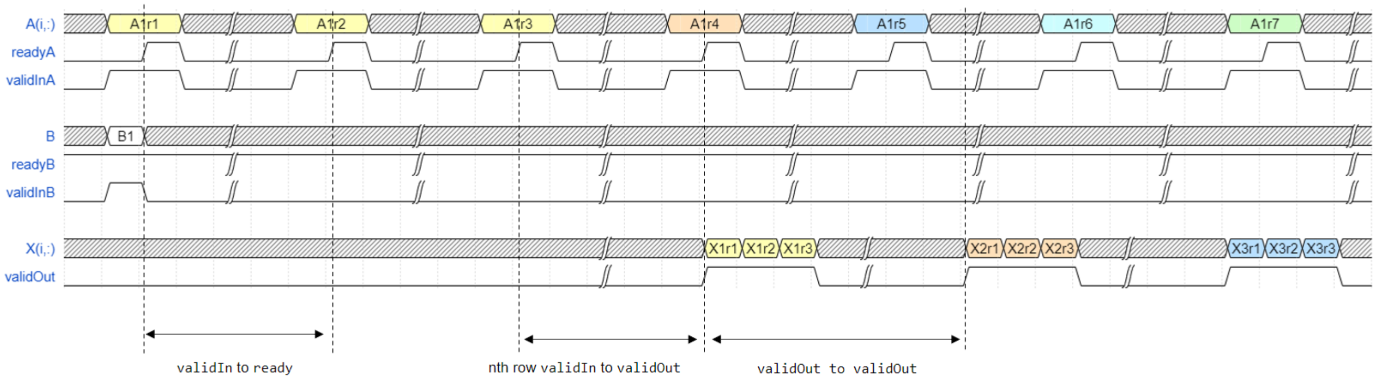
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix A row-by-row and matrix B as a single vector. After accepting the first valid pair of A and B matrices, the block outputs the X matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input A matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

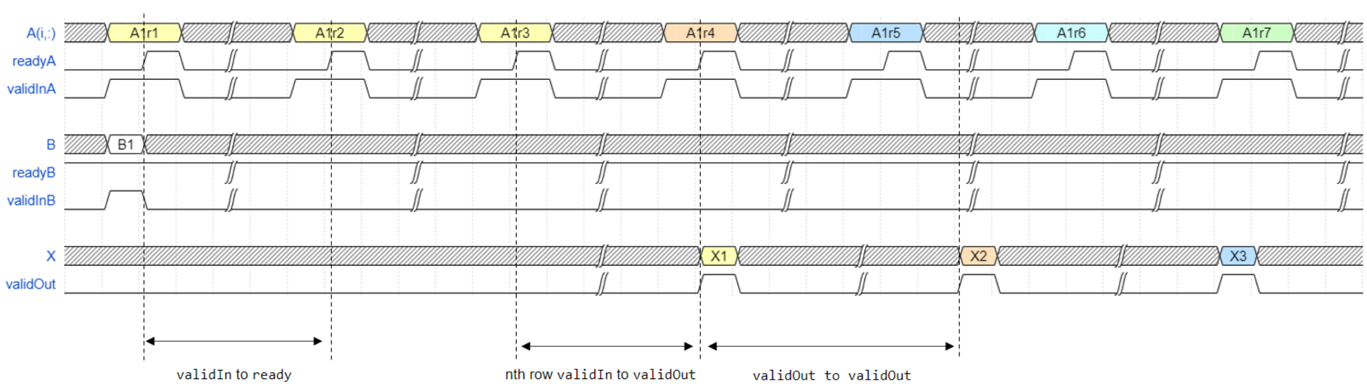


In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- n^{th} row `validIn` to `validOut` — From the n^{th} row input to the block starting to output the first solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.

- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept B matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* and *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks.

Block	Operation	`validIn` to `ready` (cycles)	`validOut` to `validOut` (cycles)	n^{th} Row `validIn` to `validOut` (cycles)
Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl + 5)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl*2 + 11)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n+ 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: inf-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	334280	425280	78.60
CLB Registers	261319	850560	30.72
DSPs	12	4272	0.28
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.892 ns
Slack	0.088 ns
Clock Frequency	255.62 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

This block depends on a partial-systolic QR decomposition block. Since 23a, when you update the diagram, the loop which composes the partial-systolic pipeline in the QR decomposition block is unrolled. This updated internal architecture removes dead operations in simulation and generated code, thus requiring fewer hardware resources. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

R2022a: Support for Tikhonov regularization parameter

The Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block now supports the Tikhonov “Regularization parameter” on page 2-0.

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

Blocks

Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Complex Partial-Systolic Q-less QR Decomposition | Complex Burst Q-less QR Decomposition

Functions

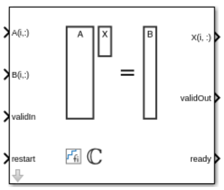
`fixed.qlessQRMatrixSolve`

Topics

"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Partial-Systolic Matrix Solve Using QR Decomposition

Compute value of x in the equation $Ax = B$ for complex-valued matrices using QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Complex Partial-Systolic Matrix Solve Using QR Decomposition block solves the system of linear equations $Ax = B$ using QR decomposition, where A and B are complex-valued matrices. To compute $x = A^{-1}$, set B to be the identity matrix.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Partial-Systolic Matrix Solve Using QR Decomposition block computes the matrix solution of complex-valued $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \begin{bmatrix} 0_{n,\,p} \\ B \end{bmatrix}$ where λ is the regularization parameter, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

B(i,:) — Rows of matrix *B*
vector

Rows of matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is `1` (`true`) and the `ready` value is `1` (`true`), the block captures the values at the `A(i,:)` and `B(i,:)` input ports. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X(i, :) — Rows of matrix *X*
scalar | vector

Rows of matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed a row of matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and the `validIn` value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in input matrices *A* and *B*
4 (default) | positive integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in input matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: positive integer-valued scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

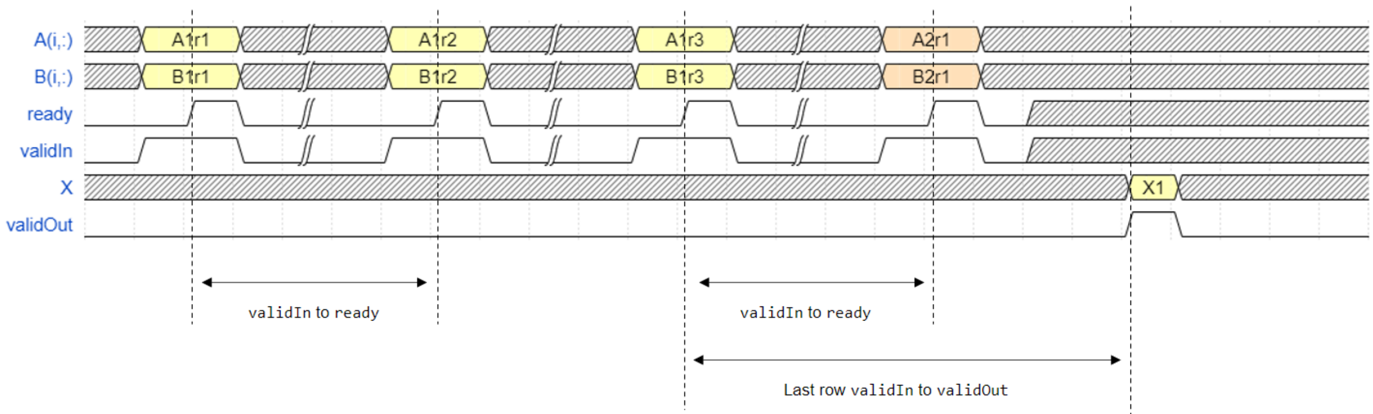
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic Matrix Solve Using QR Decomposition* blocks accept and process A and B matrices row by row. After accepting m rows, the block outputs the matrix X as a single vector. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input A and B matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix and `X1` is the matrix *X*, output as a vector.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The following table provides details of the timing for the *Partial-Systolic Matrix Solve Using QR Decomposition* blocks.

Block	**Operation**	**`validIn` to `ready` (cycles)**	**Last Row `validIn` to `validOut` (cycles)**
Real Partial-Systolic Matrix Solve Using QR Decomposition	Synchronous	max((wl+7), ceil((3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 1)/n))	(wl + 6)*n + 3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 9 - n
Complex Partial-Systolic Matrix Solve Using QR Decomposition	Synchronous	max((wl + 9), ceil((3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 1)/n))	(wl + 7.5)*2*n + 3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 9 - n

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	319045	425280	75.02
CLB Registers	261210	850560	30.71
DSPs	6	4272	0.14
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.897 ns
Slack	0.085 ns
Clock Frequency	255.43 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

This block depends on a partial-systolic QR decomposition block. Since 23a, when you update the diagram, the loop which composes the partial-systolic pipeline in the QR decomposition block is unrolled. This updated internal architecture removes dead operations in simulation and generated code, thus requiring fewer hardware resources. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

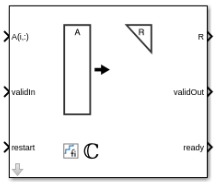
See Also
Real Partial-Systolic Matrix Solve Using QR Decomposition | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Complex Burst Matrix Solve Using QR Decomposition

Topics

"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Partial-Systolic Q-less QR Decomposition

Q-less QR decomposition for complex-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Partial-Systolic Q-less QR Decomposition block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, where A is a complex-valued matrix, without computing Q. The solution to $A'Ax = B$ is $x = R\backslash R'\backslash b$.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Partial-Systolic Q-less QR Decomposition block computes the upper-triangular factor R of the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the regularization parameter.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data at the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Matrix *R*
scalar | vector

Economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper triangular matrix. The size of matrix *R* is *n*-by-*n*. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and the `validIn` value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in input matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

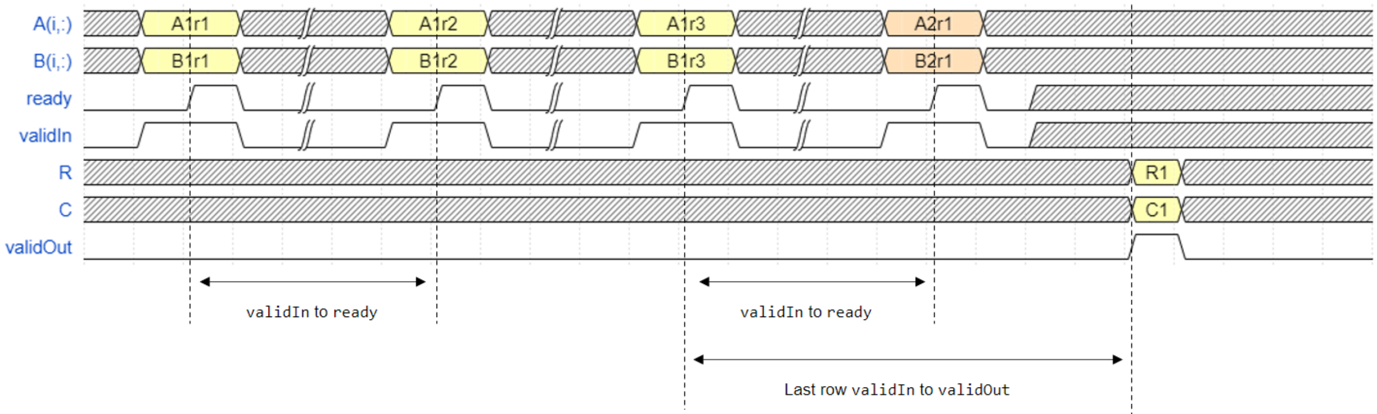
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic QR Decomposition* blocks accept and process *A* and *B* matrices row by row. After accepting *m* rows, the block outputs the *R* and *C* matrices as vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

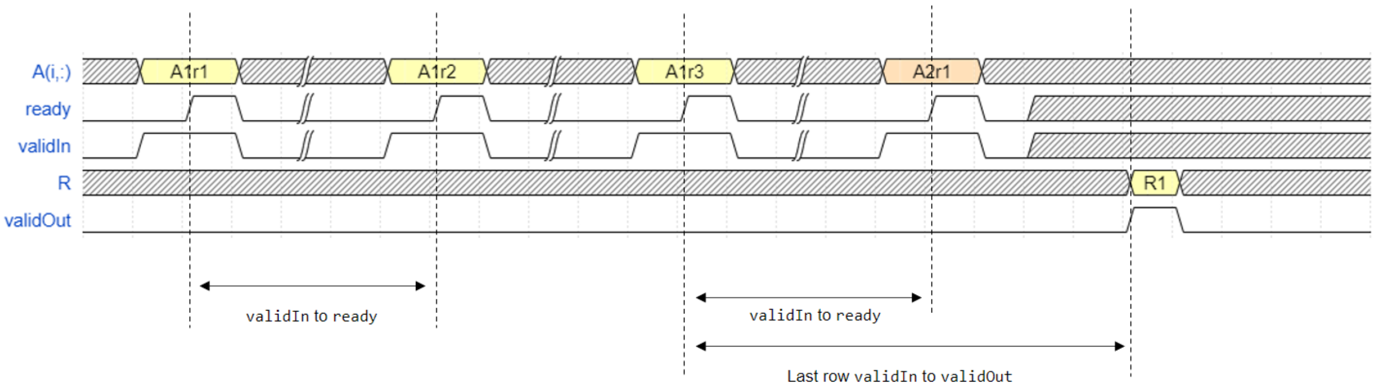


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The *Partial-Systolic Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the *R* matrices as single vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The following table provides details of the timing for the *Partial-Systolic QR Decomposition* blocks.

Block	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Partial-Systolic QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 6
Complex Partial-Systolic QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 6
Real Partial-Systolic Q-less QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 3
Complex Partial-Systolic Q-less QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 3

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The table below shows a summary of the resource utilization results.

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

Resource	Usage
LUT	91208
LUTRAM	4160
Flip Flop	57765

The tables below show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	295989	425280	69.60
CLB Registers	236040	850560	27.75
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.302 ns
Slack	0.012 ns
Clock Frequency	301.08 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

When you update the diagram, the loop which composes the partial-systolic pipeline is unrolled. This updated internal architecture removes dead operations in simulation and generated code, resulting in a significant decrease in the number of hardware resources required. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

Resource	Usage in R2022b	Usage in R2023a
LUT	160696	91208
LUTRAM	5780	4160
Flip Flop	103465	57765

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

R2022a: Support for Tikhonov regularization parameter

The Complex Partial-Systolic Q-less QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0 .

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

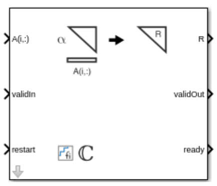
Blocks
Real Partial-Systolic Q-less QR Decomposition | Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Complex Burst Q-less QR Decomposition

Functions
`fixed.qlessQR`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor

Q-less QR decomposition for complex-valued matrices with infinite number of rows



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, without computing Q. A is an infinitely tall complex-valued matrix representing streaming data.

When the regularization parameter is nonzero, the Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor block initializes the first upper-triangular factor R to λI_n before factoring in the rows of A, where λ is the regularization parameter and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data at the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
Boolean scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (true), the block stops the current calculation and clears all internal states. When this value is 0 (false) and the validIn value is 1 (true), the block begins a new subframe.

Data Types: Boolean

Output

R — Matrix *R*
scalar | vector

Economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper triangular matrix. The size of matrix *R* is *n*-by-*n*. *R* has the same data type as *A*.

Data Types: single | double | fixed point

validOut — Whether output data is valid
Boolean scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port R is valid. When this value is 1 (true), the block has successfully computed the matrix *R*. When this value is 0 (false), the output data is not valid.

Data Types: Boolean

ready — Whether block is ready
Boolean scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (true) and the validIn value is 1 (true), the block accepts input data in the next time step. When this value is 0 (false), the block ignores input data in the next time step.

After sending a true validIn signal, there may be some delay before ready is set to false. To ensure all data is processed, you must wait until ready is set to false before sending another true validIn signal.

Data Types: Boolean

Parameters

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: n
Type: character vector
Values: positive integer-valued scalar
Default: 4

Forgetting factor — Forgetting factor applied after each row of matrix is factored
0.99 (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of A is input indefinitely.

Programmatic Use
Block Parameter: `forgettingFactor`
Type: character vector
Values: positive integer-valued scalar
Default: `0.99`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Q-less QR Decomposition with Forgetting Factor

The Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor block implements the following recursion to compute the upper-triangular factor R of continuously streaming n-by-1 row vectors $A(k,:)$ using forgetting factor α. It's as if matrix A is infinitely tall. The forgetting factor in the range $0 < \alpha < 1$ prevents it from integrating without bound.

$$\begin{aligned}
R_0 &= \text{zeros}(n, n)\\
[\sim, R_1] &= \text{qr}\left(\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix}, 0\right)\\
R_1 &= \alpha R_1\\
[\sim, R_2] &= \text{qr}\left(\begin{bmatrix} R_1 \\ A(2,:) \end{bmatrix}, 0\right)\\
R_2 &= \alpha R_2\\
&\vdots\\
[\sim, R_k] &= \text{qr}\left(\left[\begin{bmatrix} R_{k-1} \\ A(k,:) \end{bmatrix}\right], 0\right)\\
R_k &= \alpha R_k\\
&\vdots
\end{aligned}$$

Q-less QR Decomposition with Forgetting Factor and Tikhonov Regularization

The upper-triangular factor R_k after processing the k^{th} input $A(k,:)$ is computed using the following iteration.

$$
\begin{aligned}
R_0 &= \lambda I_n \\
[\sim, R_1] &= \mathrm{qr}\left(\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix}, 0\right) \\
R_1 &= \alpha R_1 \\
[\sim, R_2] &= \mathrm{qr}\left(\begin{bmatrix} R_1 \\ A(2,:) \end{bmatrix}, 0\right) \\
R_2 &= \alpha R_2 \\
&\vdots \\
[\sim, R_k] &= \mathrm{qr}\left(\begin{bmatrix} R_{k-1} \\ A(k,:) \end{bmatrix}, 0\right) \\
R_k &= \alpha R_k \\
&\vdots
\end{aligned}
$$

This is mathematically equivalent to computing the upper-triangular factor R_k of matrix A_k, defined as follows, though the block never actually creates A_k.

$$
A_k = \begin{bmatrix} \alpha^k \lambda I_n \\ \begin{bmatrix} \alpha^k & & & \\ & \alpha^{k-1} & & \\ & & \ddots & \\ & & & \alpha \end{bmatrix} A(1:k,:) \end{bmatrix}
$$

Forward and Backward Substitution

When an upper triangular factor is ready, then forward and backward substitution are computed with the current input B to produce output X.

$$X = R_k \backslash (R_k' \backslash B)$$

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

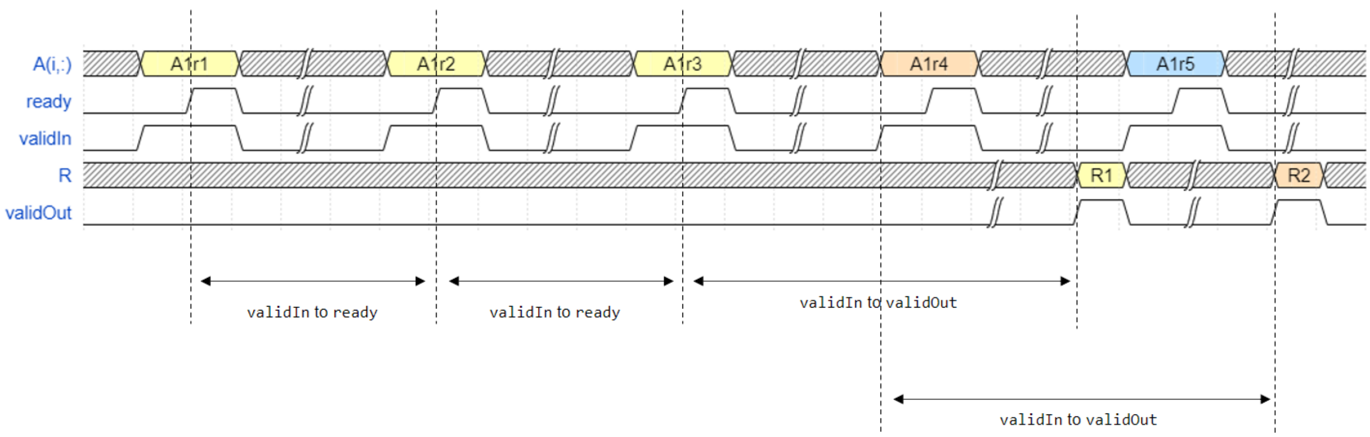
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic QR Decomposition with Forgetting Factor* blocks accept and process the matrix *A* row by row. After accepting the first *m* rows, the block starts to output the *R* matrix as a single vector. From this point, for each row input, the block calculates a *R* matrix. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input matrix *A* is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the Q-less QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- `validIn` to `validOut` — From a successful row input to the block starting to output the corresponding solution.

The following table provides details of the timing for the *Partial-Systolic Q-less QR Decomposition with Forgetting Factor* blocks.

Block	validIn to ready (cycles)	validIn to validOut (cycles)
Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor	*wl* + 7	(*wl* + 6)**n* + 3

Block	`validIn` to `ready` (cycles)	`validIn` to `validOut` (cycles)
Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor	*wl* + 9	(*wl* + 7.5)*2**n* + 3

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The table below shows a summary of the resource utilization results.

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

Resource	Usage
LUT	91496
LUTRAM	2000
Flip Flop	54609
BRAM	31

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:

  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: inf-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	327009	425280	76.89
CLB Registers	236476	850560	27.80
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.299 ns
Slack	0.016 ns
Clock Frequency	301.45 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

When you update the diagram, the loop which composes the partial-systolic pipeline is unrolled. This updated internal architecture removes dead operations in simulation and generated code, resulting in a significant decrease in the number of hardware resources required. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

Resource	R2022b	R2023a
LUT	162888	91496
LUTRAM	3620	2000
Flip Flop	100309	54609
BRAM	45	31

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:

    - m = 10
    - n = 10
    - p = 1
    - Matrix *A* dimension: 10-by-10
    - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

R2022a: Support for Tikhonov regularization parameter

The Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor block supports the Tikhonov “Regularization parameter” on page 2-0 .

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

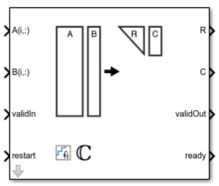
Blocks
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Complex Partial-Systolic QR Decomposition | Complex Partial-Systolic Q-less QR Decomposition | Complex Burst Q-less QR Decomposition

Functions
`fixed.qlessQR`

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Complex Partial-Systolic QR Decomposition

QR decomposition for complex-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Complex Partial-Systolic QR Decomposition block uses QR decomposition to compute R and $C = Q'B$, where $QR = A$, and A and B are complex-valued matrices. The least-squares solution to $Ax = B$ is $x = R\backslash C$. R is an upper triangular matrix and Q is an orthogonal matrix. To compute $C = Q'$, set B to be the identity matrix.

When “Regularization parameter” on page 2-0 is nonzero, the Complex Partial-Systolic QR Decomposition block transforms $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ in-place to $R = Q'\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ and $\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ in-place to $C = Q'\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ where λ is the regularization parameter, QR is the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

B(i,:) — Rows of matrix *B*
vector

Rows of matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`
Complex Number Support: Yes

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values on the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`), and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Matrix *R*
matrix

Economy-size QR decomposition matrix *R*, returned as a matrix. *R* is an upper triangular matrix. The size of matrix *R* is *n*-by-*n*. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

C — Matrix *C=Q'B*
matrix

Economy-size QR decomposition matrix *C=Q'B*, returned as a matrix or vector. *C* has the same number of rows as *R*. *C* has the same data type as *B*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at output ports `R` and `C` is valid. When this value is 1 (`true`), the block has successfully computed the *R* and *C* matrices. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`), and the `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in input matrices A and B

4 (default) | positive integer-valued scalar

The number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in input matrix A
4 (default) | positive integer-valued scalar

The number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in input matrix B
1 (default) | positive integer-valued scalar

The number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: positive integer-valued scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

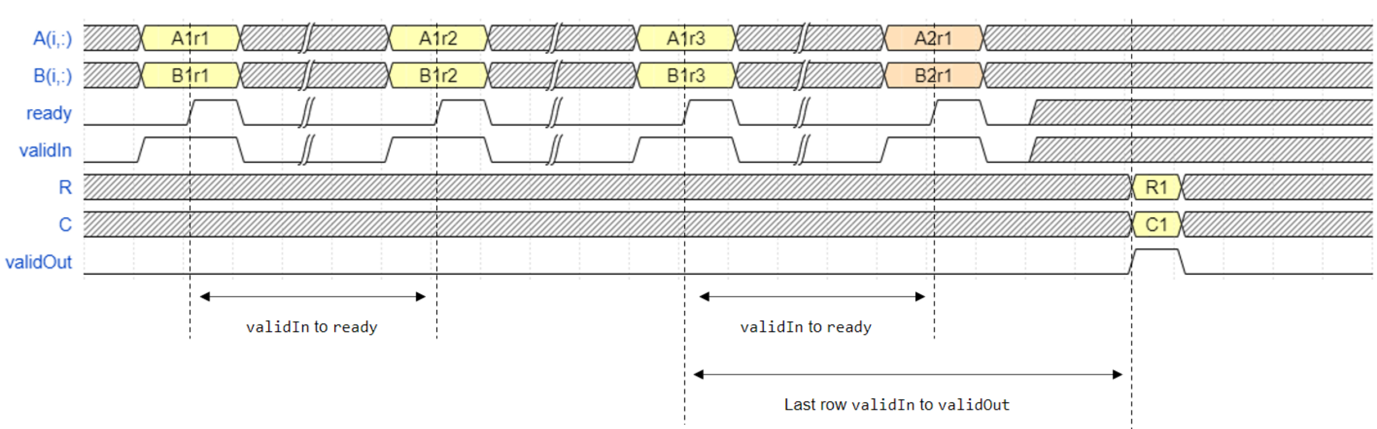
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic QR Decomposition* blocks accept and process A and B matrices row by row. After accepting m rows, the block outputs the R and C matrices as vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input A and B matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

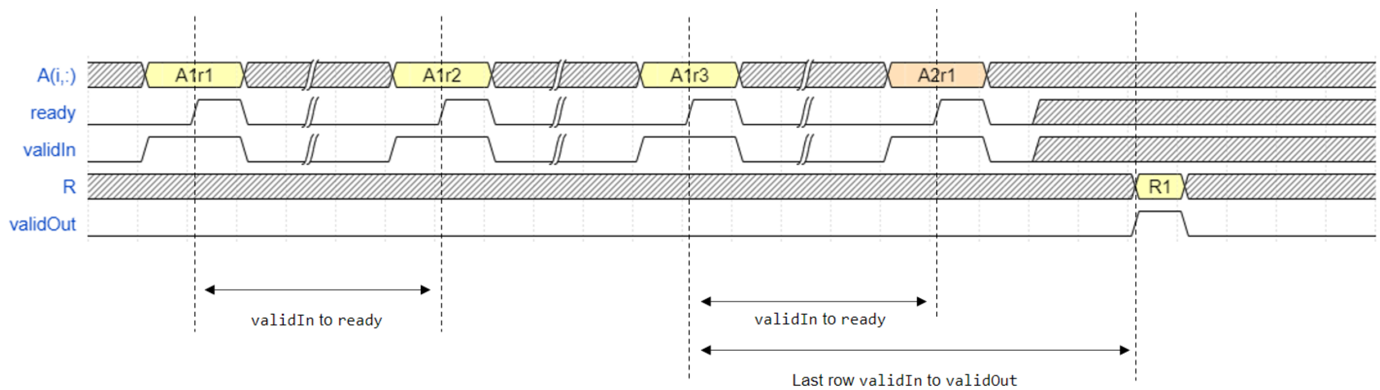


In the figure,

- A1r1 is the first row of the first *A* matrix, R1 is the first *R* matrix, and so on.
- validIn to ready — From a successful row input to the block being ready to accept the next row.
- Last row validIn to validOut — From the last row input to the block starting to output the solution.

The *Partial-Systolic Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the *R* matrices as single vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that validIn asserts before ready, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- A1r1 is the first row of the first *A* matrix, R1 is the first *R* matrix, and so on.
- validIn to ready — From a successful row input to the block being ready to accept the next row.
- Last row validIn to validOut — From the last row input to the block starting to output the solution.

The following table provides details of the timing for the *Partial-Systolic QR Decomposition* blocks.

Block	validIn to ready (cycles)	Last Row validIn to validOut (cycles)
Real Partial-Systolic QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 6
Complex Partial-Systolic QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 6
Real Partial-Systolic Q-less QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 3
Complex Partial-Systolic Q-less QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 3

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The table below shows a summary of the resource utilization results.

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

Resource	Usage
LUT	108464
LUTRAM	5000
Flip Flop	68404

The tables below show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	319908	425280	75.22
CLB Registers	250839	850560	29.49
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.299 ns
Slack	0.016 ns
Clock Frequency	301.45 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

When you update the diagram, the loop which composes the partial-systolic pipeline is unrolled. This updated internal architecture removes dead operations in simulation and generated code, resulting in a significant decrease in the number of hardware resources required. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

Resource	Usage in R2022b	Usage in R2023a
LUT	177813	108464
LUTRAM	6620	5000
Flip Flop	113857	68404

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

R2021a: Reduced HDL resource utilization

This block now has an improved algorithm to reduce resource utilization on hardware-constrained target platforms.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

Blocks

Real Partial-Systolic QR Decomposition | Complex Partial-Systolic Q-less QR Decomposition | Complex Burst QR Decomposition

Functions

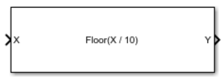
`fixed.qrAB`

Topics

"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Divide by Constant and Round

Divide input by a constant and round to integer



Libraries:
Fixed-Point Designer

Description

The Divide by Constant and Round block outputs the result of dividing the input by a constant and rounds the result to an integer using the specified rounding method.

The Divide by Constant and Round block uses an algorithm that is functionally similar to the Granlund-Montgomery-Warren Method. The division operation is computed via a multiplication by inverse, which generally results in better performance on embedded systems.

Ports

Input

X — Dividend
scalar | vector | matrix | N-D array

Dividend, specified as a scalar, vector, matrix, or N-D array.

Divide by Constant and Round does not support data types with word length greater than 128. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Output

Y — Result of division and round operation
scalar | vector | matrix | N-D array

Result of division and round operation, returned as a scalar, vector, matrix, or N-D array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

Parameters

Denominator — Divisor
10 (default) | scalar

Divisor, specified as a positive, real-valued, finite scalar.

Programmatic Use
Block Parameter: `Denominator`

Type: character vector
Values: MATLAB expression that evaluates to a positive, real-valued, finite fixed point or numeric value
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`
Default: `'10'`

Rounding Method — Rounding method to use
`Floor` (default) | `Ceiling` | `Nearest` | `Zero` | `Convergent`

Rounding method to use, specified as one of these values:

- `Floor` — Round to nearest integer in the direction of negative infinity.
- `Ceiling` — Round to nearest integer in the direction of positive infinity.
- `Nearest` — Round to the nearest integer. Ties are rounded to the nearest integer in the direction of positive infinity.
- `Zero` — Round to the nearest integer in the direction of zero.
- `Convergent` — Round to the nearest integer. Ties are rounded to the nearest even integer.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Floor'` | `'Ceiling'` | `'Nearest'` | `'Zero'` | `'Convergent'`
Default: `'Floor'`

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

See Also

Divide by Constant HDL Optimized | Divide

Topics

"Choosing a Rounding Method"

Divide by Constant HDL Optimized

Divide input by a constant and round to integer and generate optimized HDL code

Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Divide by Constant HDL Optimized block outputs the result of dividing the input by a constant and rounds the result to an integer using the specified rounding method using an HDL-optimized architecture with cycle-true latency.

The Divide by Constant HDL Optimized block uses an algorithm that is functionally similar to the Granlund-Montgomery-Warren Method. The division operation is computed via a multiplication by inverse, which generally results in better performance on embedded systems.

Ports

Input

X — Dividend
real scalar

Dividend, specified as a real scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

validIn — Whether input is valid
boolean scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the **X** input port is valid. When this value is `1` (`true`), the block captures the value on the **X** input port. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

Y — Result of division and round operation
scalar

Result of division and round operation, returned as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

validOut — Whether output data is valid
boolean scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output **Y**. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Parameters

Denominator for rational division — Divisor
10 (default) | scalar

Divisor, specified as a positive, real-valued, finite scalar.

Programmatic Use
Block Parameter: `Denominator`
Type: character vector
Values: MATLAB expression that evaluates to a positive, real-valued, finite fixed point or numeric value
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`
Default: `'10'`

Rounding Method — Rounding method to use
`Floor` (default) | `Ceiling` | `Nearest` | `Zero` | `Convergent`

Rounding method to use, specified as one of these values:

- `Floor` — Round to nearest integer in the direction of negative infinity.
- `Ceiling` — Round to nearest integer in the direction of positive infinity.
- `Nearest` — Round to the nearest integer. Ties are rounded to the nearest integer in the direction of positive infinity.
- `Zero` — Round to the nearest integer in the direction of zero.
- `Convergent` — Round to the nearest integer. Ties are rounded to the nearest even integer.

Programmatic Use
Block Parameter: `RndMeth`
Type: character vector
Values: `'Floor'` | `'Ceiling'` | `'Nearest'` | `'Zero'` | `'Convergent'`
Default: `'Floor'`

Tips

The blocks Divide by Constant HDL Optimized, Real Divide HDL Optimized, and Complex Divide HDL Optimized all perform the division operation and generate optimized HDL code.

- Real Divide HDL Optimized and Complex Divide HDL Optimized are based on a CORIDC algorithm. These blocks accept a wide variety of inputs, but will result in greater latency.
- Divide by Constant HDL Optimized accepts only real inputs and a constant divisor. Use of this block consumes DSP slices, but will complete the division operation in fewer cycles and at a higher clock rate.

Algorithms

The Divide by Constant HDL Optimized uses an HDL-optimized architecture with cycle-true latency.

The Divide by Constant HDL Optimized block uses an algorithm that is functionally similar to the Granlund-Montgomery-Warren Method. The division operation is computed via a multiplication by inverse, which generally results in better performance on embedded systems.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Slope-bias representation is not supported for fixed-point data types.

See Also

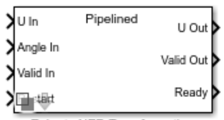
Divide by Constant and Round | Divide

Topics

"Choosing a Rounding Method"

Euler to NED Transformation HDL Optimized

Computes Euler to North-East-Down transformation using pipelined or burst architecture and generates optimized HDL code



Euler to NED Transformation HDL Optimized

Libraries:
Fixed-Point Designer HDL Support / Coordinate Transformations

Description

The Euler to NED Transformation HDL Optimized block provides two architectures that implement Euler to North-East-Down (NED) transformation using a CORDIC rotation kernel for FPGA and ASIC applications.

You can select an architecture that optimizes for either throughput or area.

- `Pipelined` — Use this architecture for high-throughput applications.
- `Burst` — Use this architecture for a minimum resource implementation.

The Euler to NED Transformation HDL Optimized block provides hardware-friendly control signals.

Ports

Input

U In — Input array
3-by-1 vector

Input array, specified as a 3-by-1 vector.

Fixed-point inputs must use binary-point scaling.

Example: `UIn = [0;0;1]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`
Complex Number Support: Yes

Angle In — Angles to rotate by
3-by-1 vector

Angles to rotate by, specified as a 3-by-1 real-valued vector containing the angles phi, theta, and psi in radians.

Example: `AngleIn = [phi;theta;psi]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Valid In — Whether input is valid
`Boolean` scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the **U In** and **Angle In** input ports are valid. When this value is `1` (`true`), the block captures the values at the input ports **U In** and **Angle In**. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the **Valid In** value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

U Out — Rotated array
3-by-1 vector

Rotated array, returned as a 3-by-1 vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fixed point`

Valid Out — Whether output data is valid
boolean scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output **U Out**. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Ready — Whether block is ready for input
`Boolean` scalar

Whether the block is ready for input, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and **Valid In** value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

Data Types: `Boolean`

Parameters

Architecture — Architecture type
`Pipelined` (default) | `Burst`

This parameter specifies the type of architecture.

- `Pipelined` — Select this value to specify low-latency architecture.

- `Burst` — Select this value to specify minimum resource architecture.

Programmatic Use
Block Parameter: `Architecture`
Type: character vector
Values: `'Pipelined'` | `'Burst'`
Default: `'Pipelined'`

Algorithms

Euler-NED Transformation

The Euler to North-East-Down (NED) transformation is carried out by the successive application of these three rotation matrices.

$$R_\phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix}$$
$$R_\theta = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$
$$R_\psi = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying these matrices together gives the total transformation.

$$A = R_\psi R_\theta R_\phi$$
$$= \begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\psi)\sin(\phi)\sin(\theta) - \cos(\phi)\sin(\psi) & \sin(\phi)\sin(\psi) + \cos(\phi)\cos(\psi)\sin(\theta) \\ \cos(\theta)\sin(\psi) & \cos(\phi)\cos(\psi) + \sin(\phi)\sin(\psi)\sin(\theta) & \cos(\phi)\sin(\psi)\sin(\theta) - \cos(\psi)\sin(\phi) \\ -\sin(\theta) & \cos(\theta)\sin(\phi) & \cos(\phi)\cos(\theta) \end{bmatrix}$$

You can transform between two frames related by the angles ϕ, θ

, and ψ by multiplying a vector in an initial frame by the matrix above.

CORDIC Algorithm

CORDIC is an acronym for COordinate Rotation Digital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations. It is an iterative algorithm that approximates the solution by converging toward the ideal point. Using CORDIC, you can calculate various functions such as sine and cosine.

To use CORDIC to solve the Euler-NED transformation, CORDIC Givens rotations are applied sequentially in the appropriate subspaces of the initial space. First, rotate by ϕ in the yz-plane, then rotate by -θ in the xz-plane, then rotate by ψ in the xy-plane.

Performance

This resource and performance data is the synthesis result from the generated HDL targeted to a Virtex®-7.

Resource Usage

Algorithm	Flip Flops	LUT	LUTRAM	DSPs
Pipelined CORDIC (sfix14En10)	3141	86	3973	0
Resource Shared CORDIC (sfix14En10)	337	659	0	0

Static Timing Analysis

Algorithm	Clock Frequency (MHz)	Latency (Cycles)	Latency (ns)
Pipelined CORDIC (sfix14En10)	347	63	181
Resource Shared CORDIC (sfix14En10)	347	57	164

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Supports fixed-point data types only. Fixed-point data types must use binary-point scaling.

Generated C/C++ code will have timing of the HDL block.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).

General	
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only. Fixed-point data types must use binary-point scaling.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

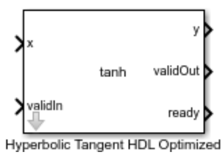
See Also

Topics

“Hardware-Efficient Euler Rotations Using CORDIC”

Hyperbolic Tangent HDL Optimized

Computes CORDIC-based hyperbolic tangent and generates optimized HDL code



Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Hyperbolic Tangent HDL Optimized block returns the hyperbolic tangent of *x*, computed using a CORDIC-based implementation optimized for HDL code generation.

Ports

Input

x — Angle in radians
real finite scalar

Angle in radians, specified as a real finite scalar. If **x** is a fixed-point or scaled double data type, **x** must use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether input is valid
`Boolean` scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the **x** input port is valid. When this value is `1` (`true`), the block captures the value on the **x** input port. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

y — Hyperbolic tangent of x
scalar

Hyperbolic tangent of the value at **x**, returned as a scalar. The value at **y** is the CORDIC-based approximation of the hyperbolic tangent of **x**. When the input to the function is floating point, the output data type is the same as the input data type. When the input is a fixed-point data type, the output has the same word length as the input and a fraction length equal to 2 less than the word length.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output **y**. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`), and the **validIn** value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

Data Types: `Boolean`

More About

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers.

The block automatically determines the number of iterations, `niters`, the CORDIC algorithm performs based on the data type of the input.

Data type of input x	`niters`
single	23
double	52
fixed point	One less than the word length of **x**. The minimum number of CORDIC iterations is 7.

Hardware Efficient Fixed-Point Computations

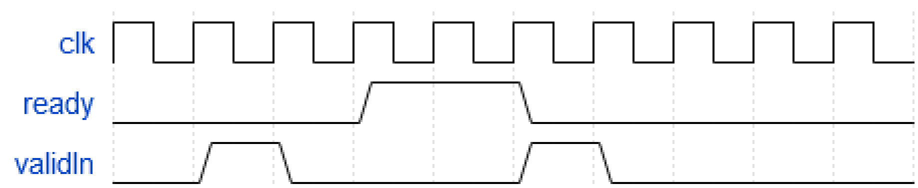
The Hyperbolic Tangent HDL Optimized block supports HDL code generation for fixed-point data with binary-point scaling. It is designed with this application in mind, and employs hardware specific semantics and optimizations. One of these optimizations is resource sharing.

When deploying intricate algorithms to FPGA or ASIC devices, there is often a trade-off between resource usage and total throughput for a given computation. Fully pipelined and parallelized algorithms have the greatest throughput, but they are often too resource intensive to deploy on real devices. By implementing scheduling logic around one or several core computational circuits, it is possible to reuse resources throughout a computation. The result is an implementation with a much smaller footprint, at the cost of a reduced total throughput. This is often an acceptable trade-off, as resource shared designs can still meet overall latency requirements.

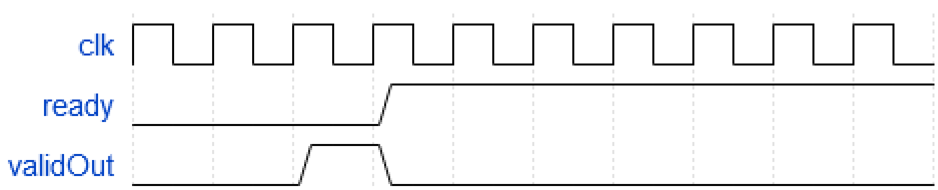
All of the key computational units in the Hyperbolic Tangent HDL Optimized block are reused throughout the computation life cycle. This includes not only the CORDIC circuitry used to perform the Givens rotations, but also the adders and multipliers used for updating the angles. This saves both DSP and fabric resources when deploying to FPGA or ASIC devices.

How to Interface with the Hyperbolic Tangent HDL Optimized Block

The Hyperbolic Tangent HDL Optimized block accepts data when the **ready** output is high, indicating that the block is ready to begin a new computation. To send input data to the block, the **validIn** signal must be asserted. If the block successfully registers the input value it will de-assert the ready signal, and the user must then wait until the signal is asserted again to send a new input. This protocol is summarized in the following wave diagram. Note how the first valid input to the block is discarded because the block was not ready to accept input data.



When the block has finished the computation and is ready to send the output, it will assert **validOut** for one clock cycle. Then **ready** will be asserted, indicating that the block is ready to accept a new input value.



Version History
Introduced in R2020a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

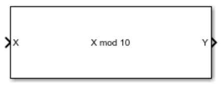
See Also

Functions

`cordictanh`

Modulo by Constant

Perform modulo operation with a constant denominator



Libraries:
Fixed-Point Designer

Description

The Modulo by Constant block performs the modulo operation (remainder after division) with a constant denominator.

The Modulo by Constant block uses an algorithm that is functionally similar to a Barrett Reduction. The division operation is computed via a multiplication by inverse, which generally results in better performance on embedded systems.

Ports

Input

X — Dividend
real scalar

Dividend, specified as a real scalar.

If X is a fixed-point data type, it must use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

Output

Y — Result of modulus operation
scalar

Result of modulus operation, returned as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

Parameters

Denominator for Modulo Problem — Divisor
10 (default) | scalar

Divisor to use for the modulus operation, specified as a positive, real-valued, finite scalar.

Programmatic Use
Block Parameter: `Denominator`

Type: character vector
Values: MATLAB expression that evaluates to a positive, real-valued, finite fixed point or numeric value
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`
Default: `'10'`

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

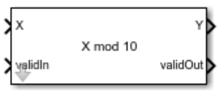
Slope-bias representation is not supported for fixed-point data types.

See Also

Modulo by Constant HDL Optimized

Modulo by Constant HDL Optimized

Perform mod operation with a constant denominator and generate optimized HDL code



Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Modulo by Constant HDL Optimized block performs the modulo operation (remainder after division) with a constant denominator using an HDL-optimized architecture with cycle-true latency.

The Modulo by Constant block uses an algorithm that is functionally similar to a Barrett Reduction. The division operation is computed via a multiplication by inverse, which generally results in better performance on embedded systems.

Ports

Input

X — Dividend
real scalar

Dividend, specified as a real scalar.

If **X** is a fixed-point data type, it must use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

validIn — Whether input is valid
boolean scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the **X** input port is valid. When this value is `1` (`true`), the block captures the value on the **X** input port. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

Y — Result of modulus operation
scalar

Result of modulus operation, returned as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `fixed point`

validOut — Whether output data is valid
boolean scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output **Y**. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Parameters

Denominator — Divisor
10 (default) | real scalar

Divisor to use for the modulus operation, specified as a positive, real-valued, finite scalar.

Programmatic Use
Block Parameter: `Denominator`
Type: character vector
Values: MATLAB expression that evaluates to a positive, real-valued, finite fixed point or numeric value
Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean` | `fixed point`
Default: `'10'`

Algorithms

The Modulo by Constant HDL Optimized block performs the modulo operation (remainder after division) with a constant denominator using an HDL-optimized architecture with cycle-true latency.

The modulo operation,

$$Y = X\text{mod}D = X - \left\lfloor \frac{X}{D} \right\rfloor \times D$$

is an important building block for many mathematical algorithms. However, this formula for X mod D is computationally inefficient for fixed-point and integer inputs. Many embedded processors lack instructions for integer division. Those that do have them require many clock cycles to compute the answer. Division is also inefficient in commercially-available FPGAs, whose arithmetic circuits are designed for efficient multiplication, addition, and subtraction. Finally, for fixed-point modulo operations, it is difficult to optimize the word length of internal data types used for the calculation because the division operation is unbounded, even for small-wordlength inputs.

The denominator in the modulo problem is a compile-time constant, so the block can compute the floored division by using a multiplication followed by a cast. Rewriting the division operation as

$$\frac{X}{D} = X \times \frac{1}{D}$$

shows this. The constant is calculated to the precision necessary to maintain both accuracy and computational efficiency. The cast that follows discards any fractional bits, which is an efficient operation on both microprocessors and FPGAs.

The Modulo by Constant block uses an algorithm that is functionally similar to a Barrett Reduction. The division operation is computed via a multiplication by inverse, which generally results in better performance on embedded systems.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

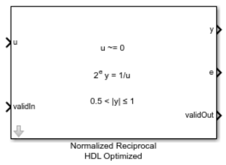
Slope-bias representation is not supported for fixed-point data types.

See Also

Modulo by Constant

Normalized Reciprocal HDL Optimized

Computes normalized reciprocal and generates optimized HDL code



Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Normalized Reciprocal HDL Optimized block computes the normalized reciprocal of u, returned as y and t such that $0.5 < |y| \leq 1$ and $2^e y = 1/u$.

- If $u = 0$ and u is a fixed-point or scaled-double data type, then $y = 2 - \text{eps}(y)$ and $e = 2^{\text{nextpow2}(w)} - w + f$, where `w` is the word length of u and f is the fraction length of u.
- If $u = 0$ and u is a floating-point data type, then $y =$ `Inf` and $t = 1$.

Ports

Input

u — Value to take normalized reciprocal of
real scalar

Value to take the normalized reciprocal of, specified as a real scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether input is valid
`Boolean` scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the **u** input port is valid. When this value is `1` (`true`), the block captures the value at the **u** input port. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

y — Normalized reciprocal
scalar

Normalized reciprocal that satisfies $0.5 < |y| \leq 1$ and $2^e y = 1/u$, returned as a scalar.

- If the input at port **u** is a signed fixed-point or scaled-double data type with word length w, then **y** is a signed fixed-point or scaled-double data type with word length w and fraction length $w - 2$.

- If the input at port **u** is an unsigned fixed-point or scaled-double data type with word length *w*, then **y** is an unsigned fixed-point or scaled-double data type with word length *w* and fraction length *w* – 1.
- If the input at port **u** is a double, then **y** is a double.
- If the input at port **u** is a single, the **y** is a single.

Data Types: `single` | `double` | `fixed point`

e — Exponent
integer scalar

Exponent that satisfies $0.5 < |y| \le 1$ and $2^e y = 1/u$, returned as an integer scalar.

Data Types: `int32`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the outputs at ports **y** and **e**. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Algorithms

The Normalized Reciprocal HDL Optimized block works by normalizing the input using a binary search, which has a latency of approximately log2 of the word length of the input, followed by a CORDIC reciprocal kernel, which has a latency approximately the same as the word length of the input.

The Normalized Reciprocal HDL Optimized block is always ready to accept data. After the initial latency, valid samples are output every sample. The latency in samples for a fixed-point input **u** is

```
D = ceil(log2(u.WordLength)) + u.WordLength + 5
```

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

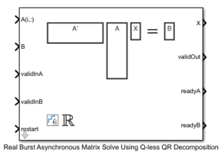
See Also

Functions
`normalizedReciprocal`

Blocks
HDL Reciprocal

Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition

Compute the value of X in the equation $A'AX = B$ for real-valued matrices using asynchronous Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition block solves the system of linear equations $A'AX = B$ using asynchronous Q-less QR decomposition, where A and B are real-valued matrices.

When "Regularization parameter" on page 2-0 is nonzero, the Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

This block operates asynchronously. The forward- and backward-substitution and Q-less QR decomposition run independently using the latest R and B matrices.

Ports

Input

A(i,:) — Rows of real matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B — Rows of real matrix *B*
vector

Rows of real matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether `A(i,:)` input is valid
`Boolean` scalar

Whether `A(i,:)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (`true`), the block captures the values at the `A(i,:)` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

validInB — Whether `B` input is valid
`Boolean` scalar

Whether `B` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `B` input port is valid. When this value is `1` (`true`) and the `readyB` value is `1` (`true`), the block captures the values at the `B` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `X` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

readyA — Whether block is ready for input `A(i,:)`
`Boolean` scalar

Whether block is ready for input `A(i,:)`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validInA` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input `B`
`Boolean` scalar

Whether block is ready for input `B`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validInB` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
4 (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix B — Number of columns in matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

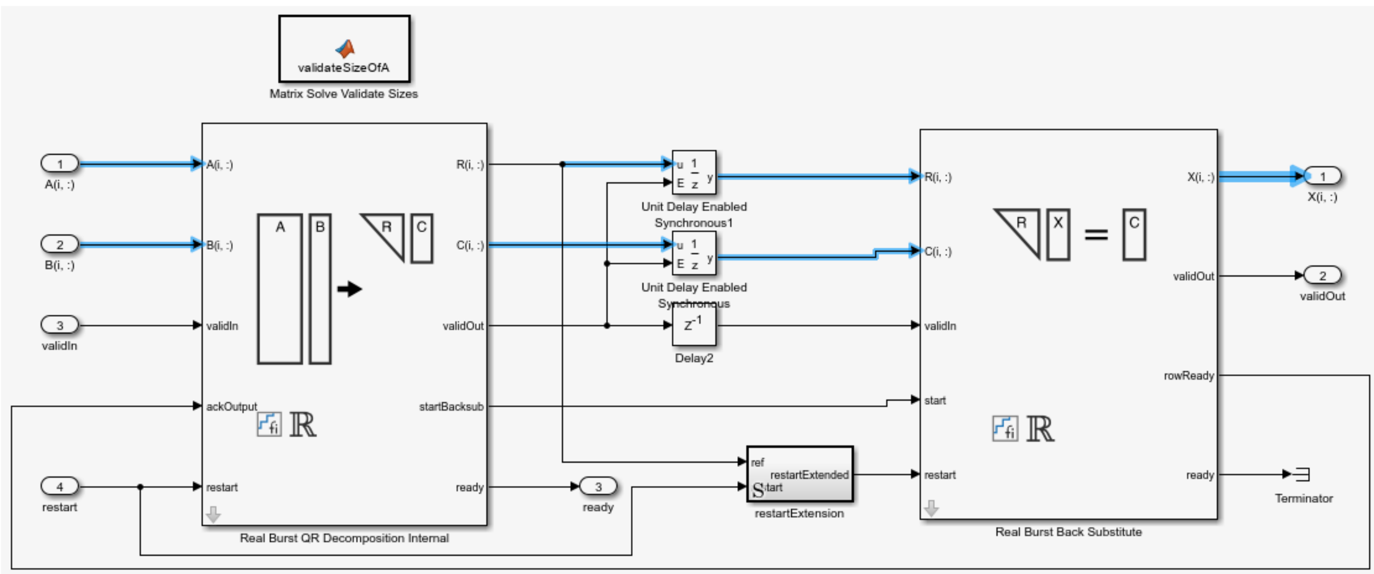
Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

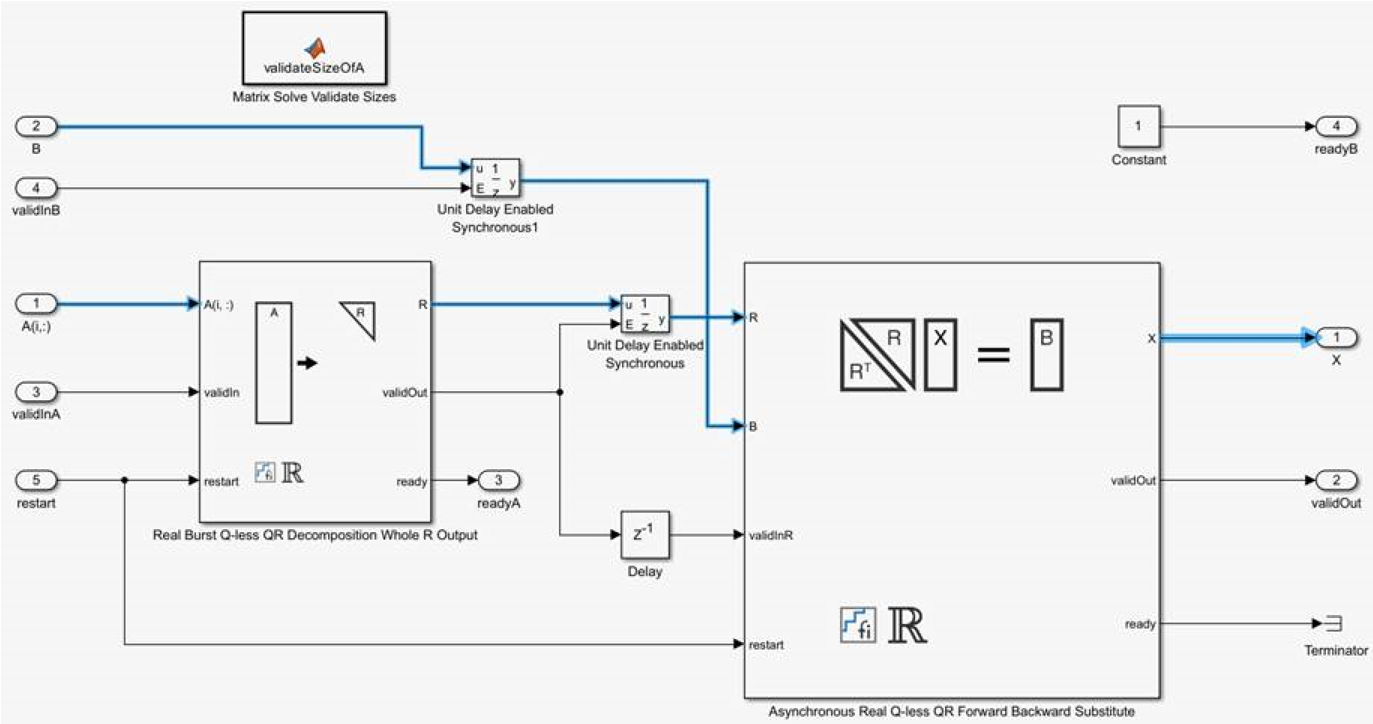
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input A matrix and the resulting R matrix is put into a buffer. Then, a forward backward substitution block uses the input B matrix and the buffered R matrix to compute $R'RX = B$. Because the R and B matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first R and B matrices are available. Then it runs continuously using the latest buffered R and B matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing A and B matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of R and B matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

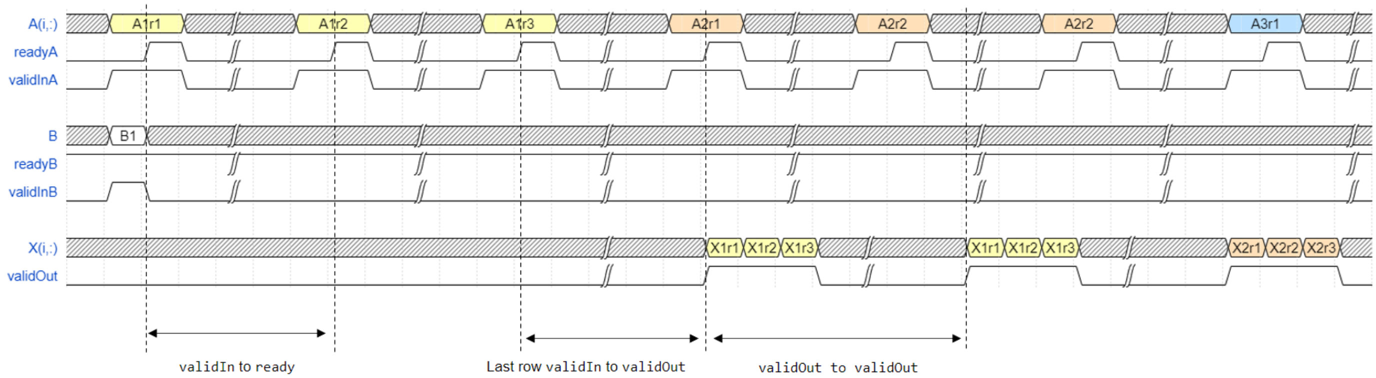
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Asynchronous Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

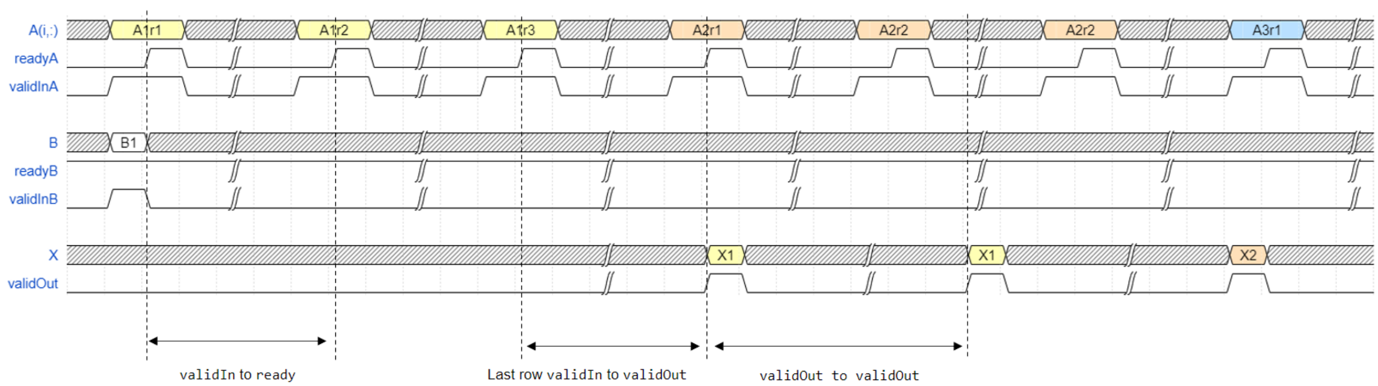


In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.

- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept B matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	`validOut` to `validOut` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl+5)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl*2 + 11)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see "HDL Code Generation and FPGA Synthesis from Simulink Model" (HDL Coder) and "Implement Digital Downconverter for FPGA" (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:

  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	16131	425280	3.79
CLB Registers	21469	850560	2.52
DSPs	4	4272	0.09
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.544 ns
Slack	0.437 ns
Clock Frequency	280.66 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

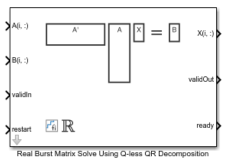
See Also

Blocks
Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition | Real Burst Matrix Solve Using Q-less QR Decomposition | Real Burst Matrix Solve Using QR Decomposition

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst Matrix Solve Using Q-less QR Decomposition

Compute the value of X in the equation $A'AX = B$ for real-valued matrices using Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Burst Matrix Solve Using Q-less QR Decomposition block solves the system of linear equations $A'AX = B$ using Q-less QR decomposition, where A and B are real-valued matrices.

When "Regularization parameter" on page 2-0 is nonzero, the Real Burst Matrix Solve Using Q-less QR Decomposition block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of real matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of real matrix *B*
vector

Rows of real matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is `1` (`true`) and the value at `ready` is `1` (`true`), the block captures the values at the `A(i,:)` and `B(i,:)` input ports. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X(i,:) — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X(i,:)` is valid. When this value is `1` (`true`), the block has successfully computed a row of *X*. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and the `validIn` value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
`4` (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector

Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Tips

Use `fixed.getQlessQRMatrixSolveModel(A,B)` to generate a template model containing a Real Burst Matrix Solve Using Q-less QR Decomposition block for real-valued input matrices `A` and `B`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

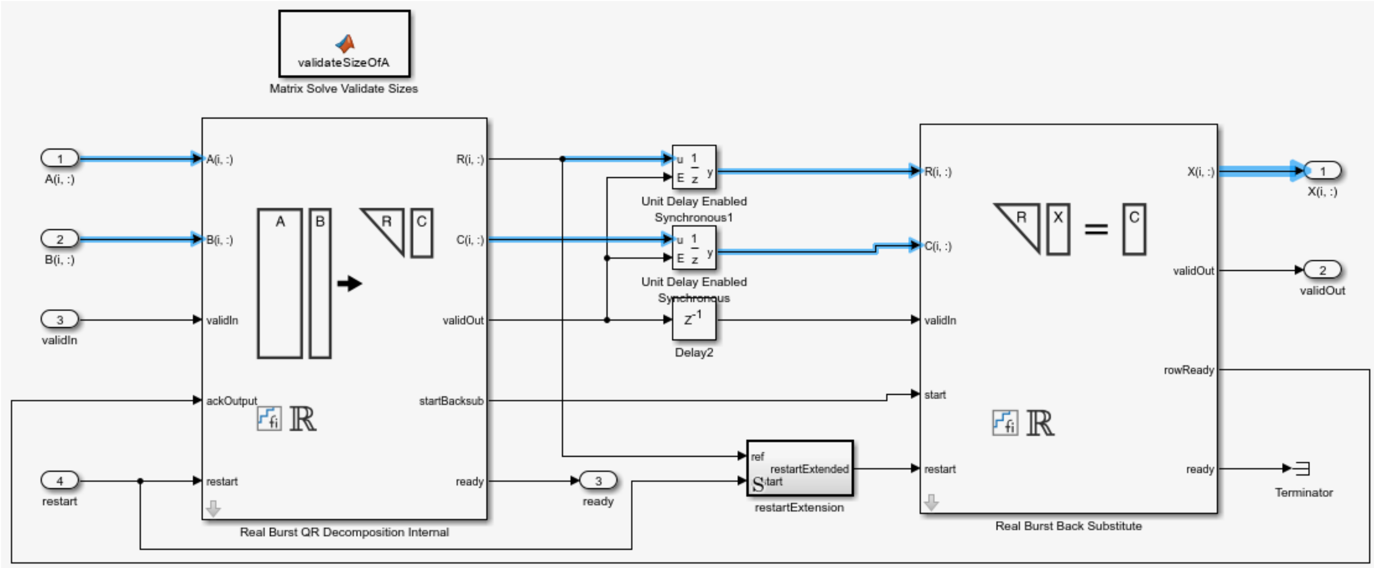
Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

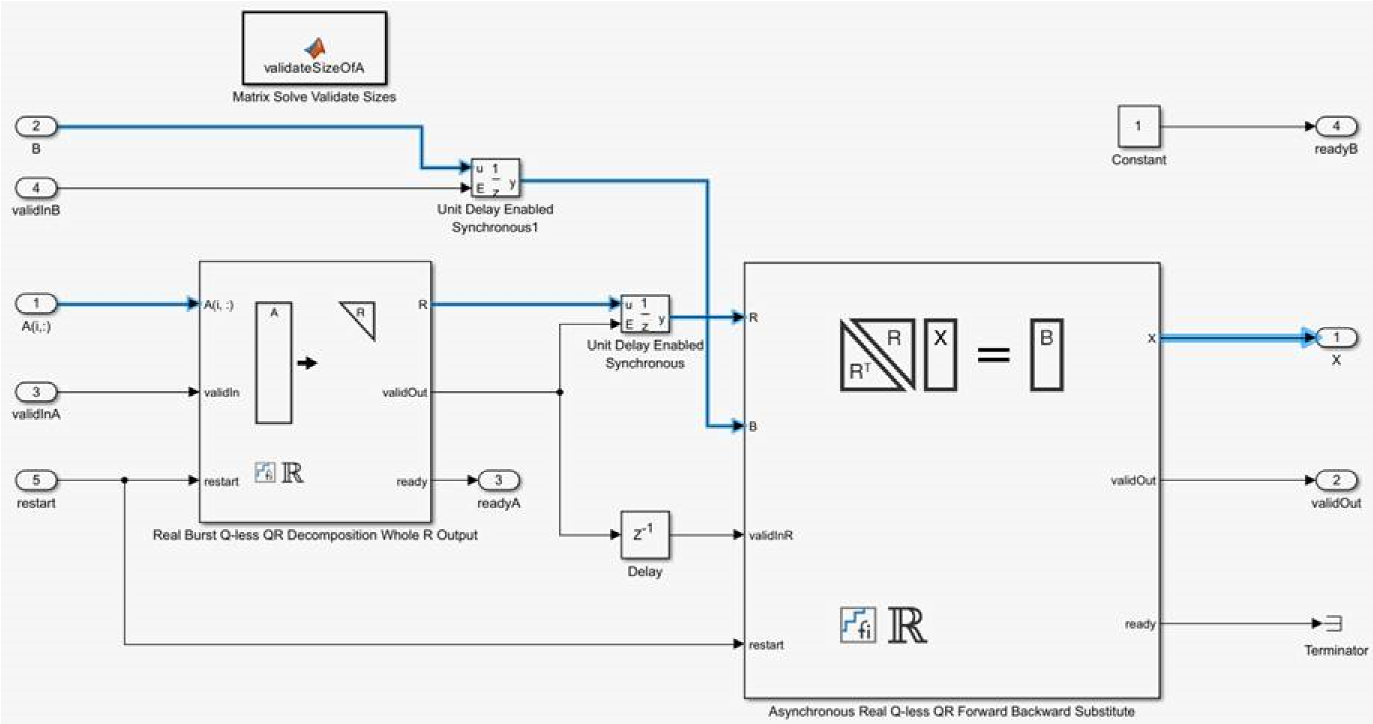
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input *A* matrix and the resulting *R* matrix is put into a buffer. Then, a forward backward substitution block uses the input *B* matrix and the buffered *R* matrix to compute $R'RX = B$. Because the *R* and *B* matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first *R* and *B* matrices are available. Then it runs continuously using the latest buffered *R* and *B* matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing *A* and *B* matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of *R* and *B* matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

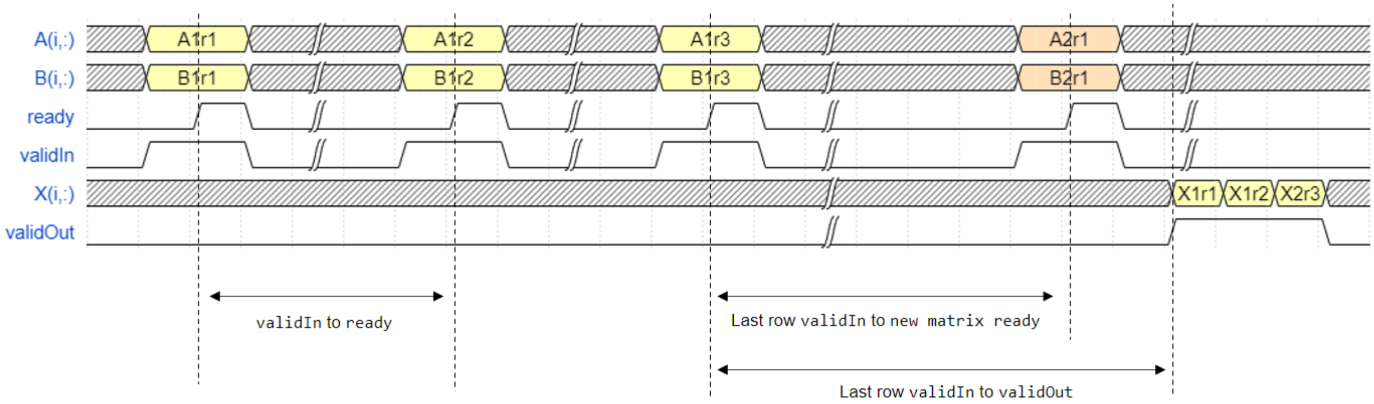
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

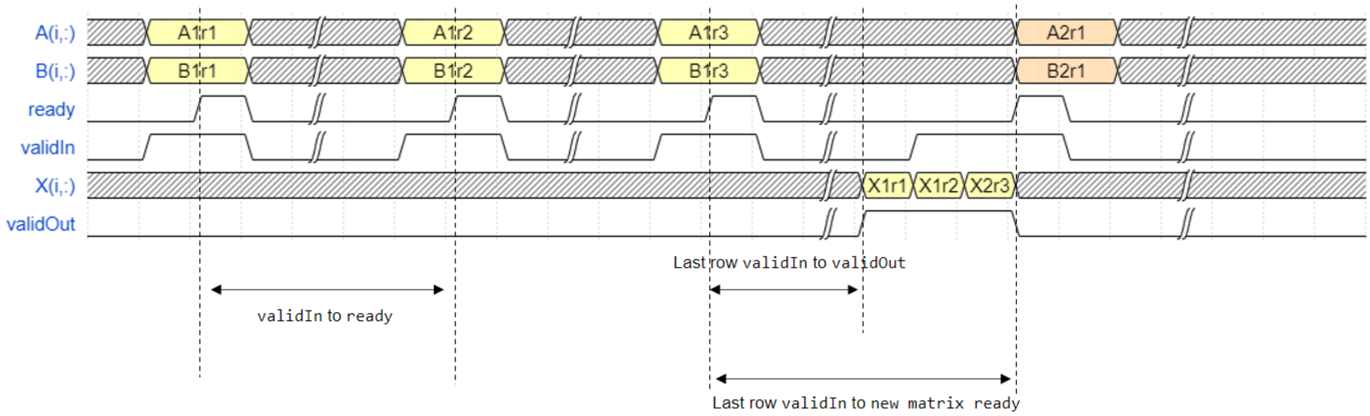


In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Matrix Solve Using Q-less QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)	Last row `validIn` to new matrix ready (cycles)
Real Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$(wl + 5)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl + 5)*n + 3.5*(n - 1)^2 + (n - 1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Complex Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$(wl*2 + 11)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl*2 + 11)*n + 3.5*(n-1)^2 + (n-1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Real Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$
Complex Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$7*2^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`

  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	13653	425280	3.21
CLB Registers	15739	850560	1.85
DSPs	4	4272	0.09
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.468 ns
Slack	0.427 ns
Clock Frequency	279.88 MHz

Version History

Introduced in R2020a

R2022a: Support for Tikhonov regularization parameter

The Real Burst Matrix Solve Using Q-less QR Decomposition block now supports the Tikhonov "Regularization parameter" on page 2-0.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

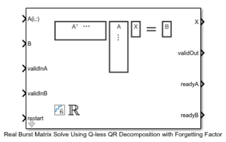
See Also

Blocks
Complex Burst Matrix Solve Using Q-less QR Decomposition | Real Burst Matrix Solve Using QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Compute the value of X in the equation $A'AX = B$ for real-valued matrices with infinite number of rows using asynchronous Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block solves the system of linear equations $A'AX = B$ using asynchronous Q-less QR decomposition, where A and B are real-valued matrices. A is an infinitely tall matrix representing streaming data.

When “Regularization parameter” on page 2-0 is nonzero, the Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

This block operates asynchronously. The forward- and backward-substitution and Q-less QR decomposition run independently using the latest R and B matrices.

Ports

Input

A(i,:) — Rows of real matrix A
vector

Rows of real matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B — Rows of real matrix B
vector

Rows of real matrix B, specified as a vector. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether `A(i,:)` input is valid
`Boolean` scalar

Whether `A(i,:)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (`true`), the block captures the values at the `A(i,:)` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

validInB — Whether `B` input is valid
`Boolean` scalar

Whether `B` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `B` input port is valid. When this value is `1` (`true`) and the `readyB` value is `1` (`true`), the block captures the values at the `B` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `X` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

readyA — Whether block is ready for input `A(i,:)`
`Boolean` scalar

Whether block is ready for input `A(i,:)`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validInA` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input `B`
`Boolean` scalar

Whether block is ready for input `B`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validInB` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
`4` (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Forgetting factor — Forgetting factor applied after each row of the matrix is factored
`0.99` (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of A is input indefinitely.

Programmatic Use
Block Parameter: `forgetting_factor`
Type: character vector
Values: real positive scalar
Default: `0`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix X
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix X, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

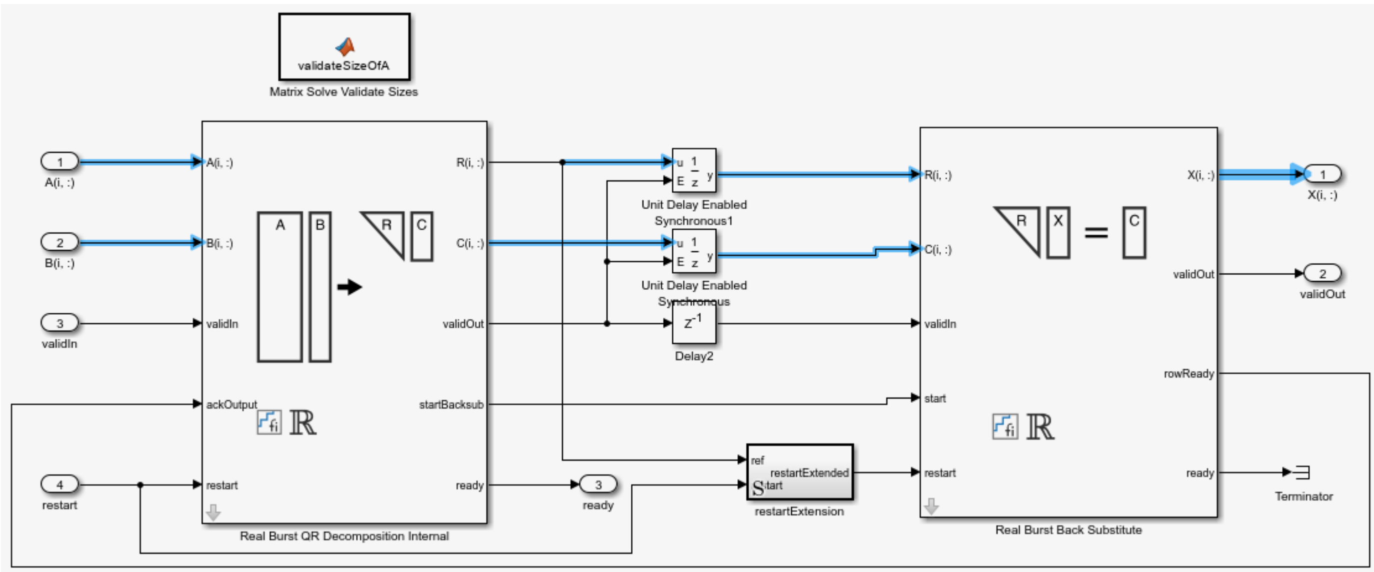
Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

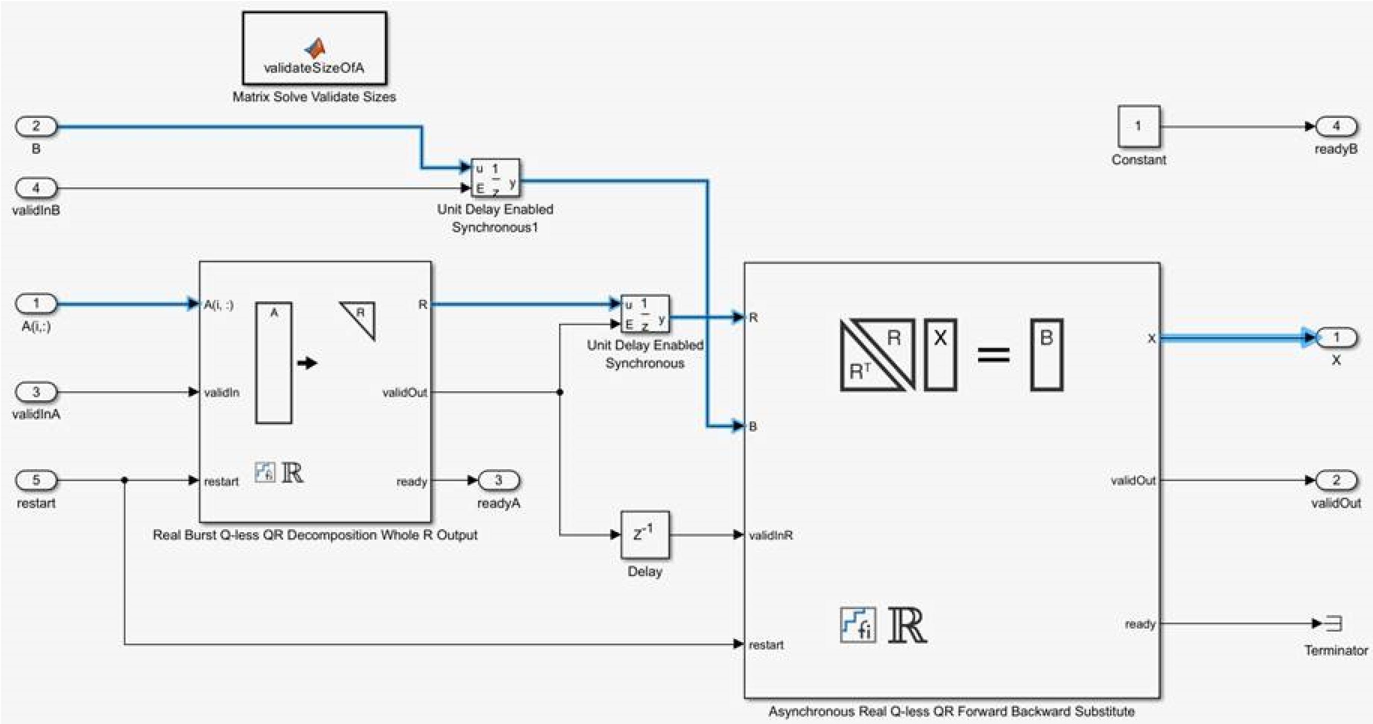
For additional considerations in selecting a block for your application, see "Choose a Block for HDL-Optimized Fixed-Point Matrix Operations".

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input A matrix and the resulting R matrix is put into a buffer. Then, a forward backward substitution block uses the input B matrix and the buffered R matrix to compute $R'RX = B$. Because the R and B matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first R and B matrices are available. Then it runs continuously using the latest buffered R and B matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing A and B matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of R and B matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

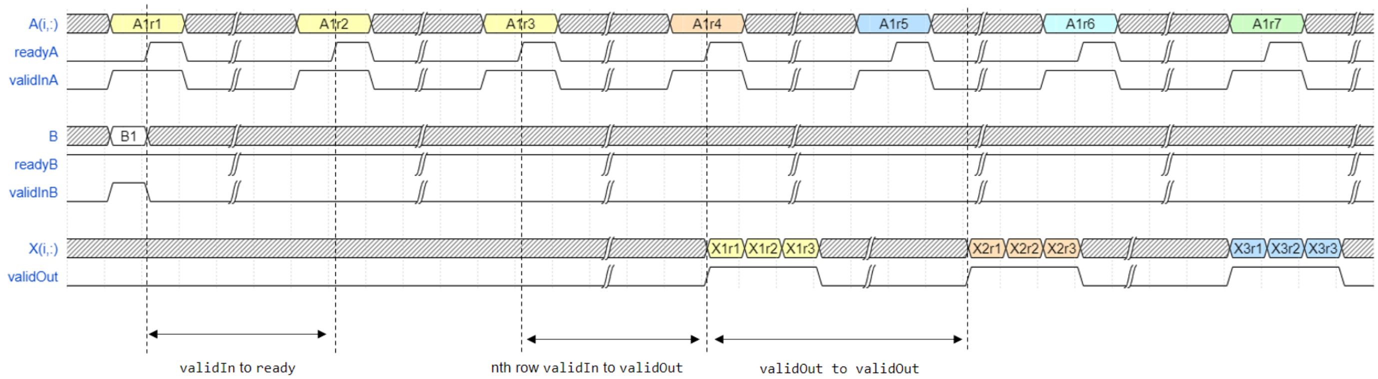
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

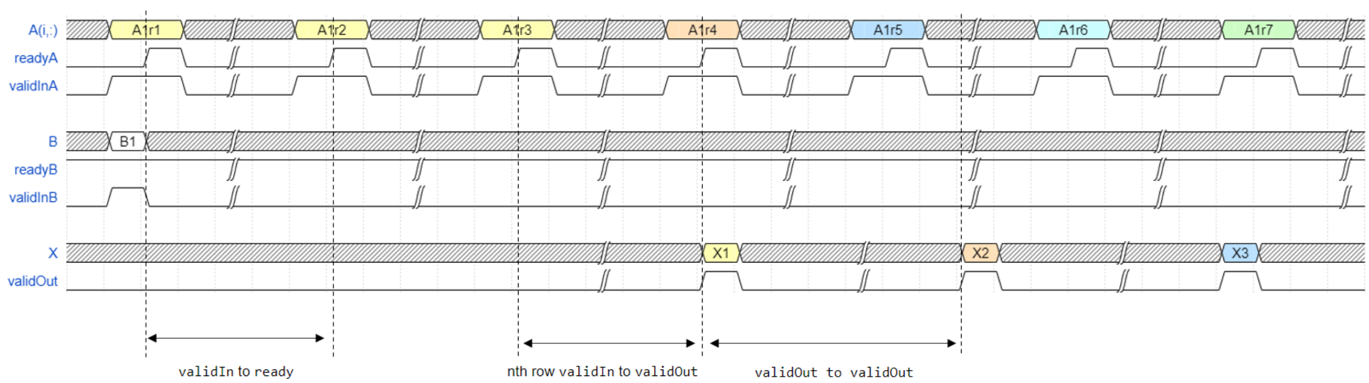


In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- n^{th} row `validIn` to `validOut` — From the n^{th} row input to the block starting to output the first solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.

- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept B matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* and *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks.

Block	**Operation**	**`validIn` to `ready` (cycles)**	**`validOut` to `validOut` (cycles)**	**n^{th} Row `validIn` to `validOut` (cycles)**
Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl + 5)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl*2 + 11)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n+ 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: inf-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	16148	425280	3.80
CLB Registers	21484	850560	2.53
DSPs	20	4272	0.47
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.084 ns
Slack	0.23 ns
Clock Frequency	322.23 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

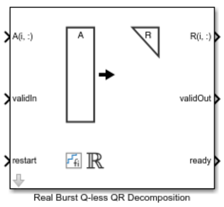
See Also

Blocks
Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst Q-less QR Decomposition

Q-less QR decomposition for real-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Burst Q-less QR Decomposition block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, where A is a real-valued matrix, without computing Q. The solution to $A'Ax = B$ is $x = R\backslash R'\backslash b$.

When “Regularization parameter” on page 2-0 is nonzero, the Real Burst Q-less QR Decomposition block computes the upper-triangular factor R of the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the regularization parameter.

Ports

Input

A(i,:) — Rows of real matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If A is a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R(i,:) — Rows of upper-triangular matrix *R*
scalar | vector

Rows of the economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper triangular matrix. The size of the matrix *R* is min(*m*,*n*)-by-*n*. The output at `R(i,:)` has the same data type as the input at `A(i,:)`.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A — Number of columns in matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: n
Type: character vector
Values: positive integer-valued scalar
Default: 4

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: 0

Tips

Use `fixed.getQlessQRDecompositionModel(A)` to generate a template model containing a Real Burst Q-less QR Decomposition block for real-valued input matrix `A`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

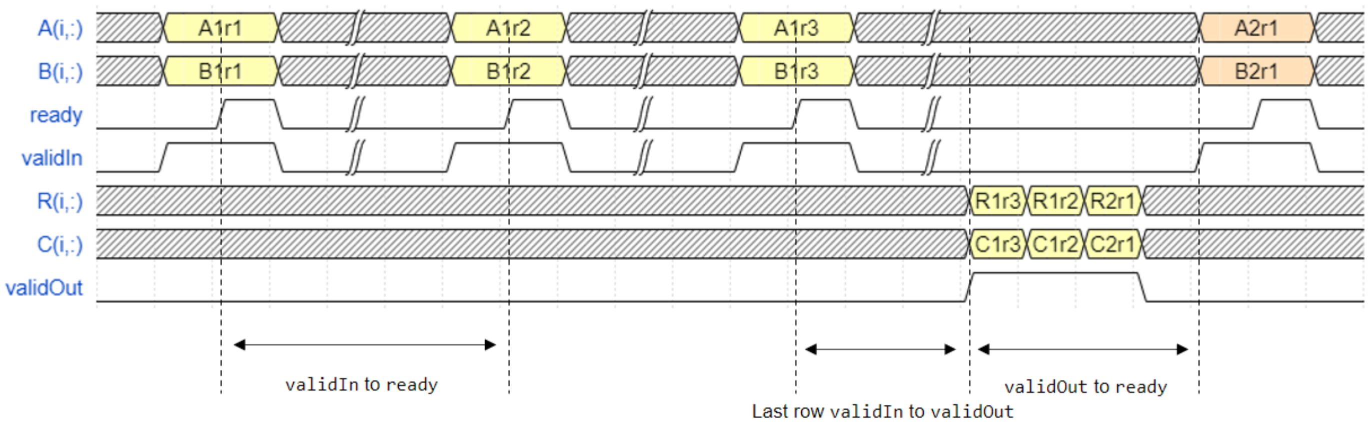
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *R* and *C* matrices row by row continuously. The matrices are output from the last row to the first row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

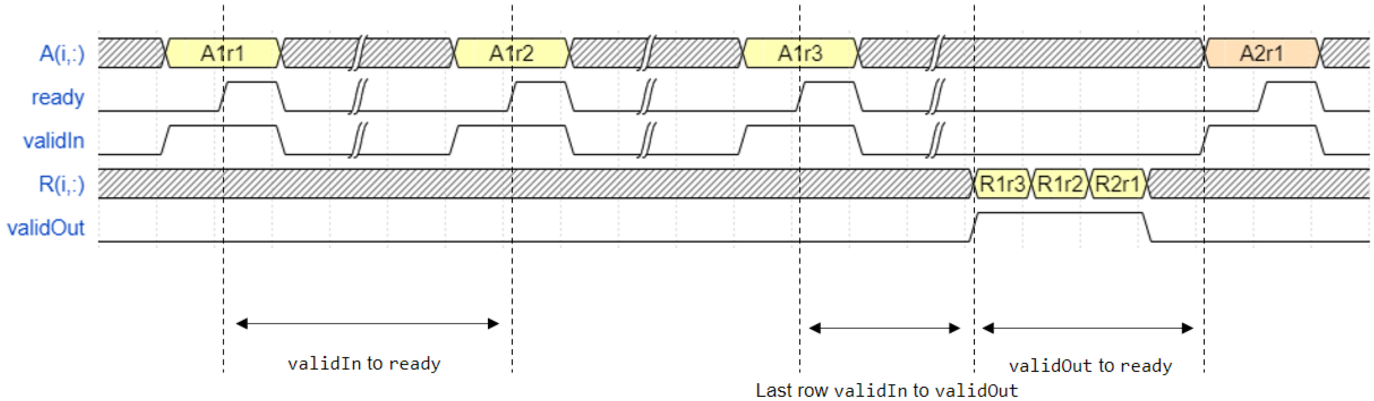


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the matrix *R* row by row continuously. The matrix is output from the last row to the first row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst QR Decomposition* blocks.

Block	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)	`validOut` to `ready` (cycles)
Real Burst QR Decomposition	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2$	$\min(m,n) + 1$
Complex Burst QR Decomposition	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2$	$\min(m,n) + 1$
Real Burst Q-less QR Decomposition	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2$	$\min(m,n) + 1$
Complex Burst Q-less QR Decomposition	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2$	$\min(m,n) + 1$

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - Matrix *A* dimension: 16-by-16
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	8006	425280	1.88
CLB Registers	8286	850560	0.97
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	2.89 ns
Slack	0.338 ns
Clock Frequency	333.85 MHz

Version History

Introduced in R2020a

R2022a: Support for Tikhonov regularization parameter

The Real Burst Q-less QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

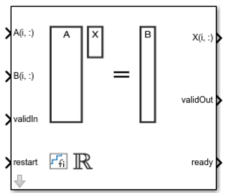
Blocks
Real Burst QR Decomposition | Complex Burst Q-less QR Decomposition | Real Partial-Systolic QR Decomposition

Functions
`fixed.qlessQR`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst Matrix Solve Using QR Decomposition

Compute the value of x in the equation $Ax = B$ for real-valued matrices using QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Burst Matrix Solve Using QR Decomposition block solves the system of linear equations $Ax = B$ using QR decomposition, where A and B are real-valued matrices. To compute $x = A^{-1}$, set B to be the identity matrix.

When "Regularization parameter" on page 2-0 is nonzero, the Real Burst Matrix Solve Using QR Decomposition block computes the matrix solution of real-valued $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ where λ is the regularization parameter, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of real matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of real matrix *B*
vector

Rows of real matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values on the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X(i,:) — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed a row of matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and the `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in matrices *A* and *B*
`4` (default) | positive integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in matrix *A*
`4` (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: positive integer-valued scalar
Default: `0`

Output datatype — Data type of the output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Tips

Use `fixed.getMatrixSolveModel(A,B)` to generate a template model containing a Real Burst Matrix Solve Using QR Decomposition block for real-valued input matrices `A` and `B`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

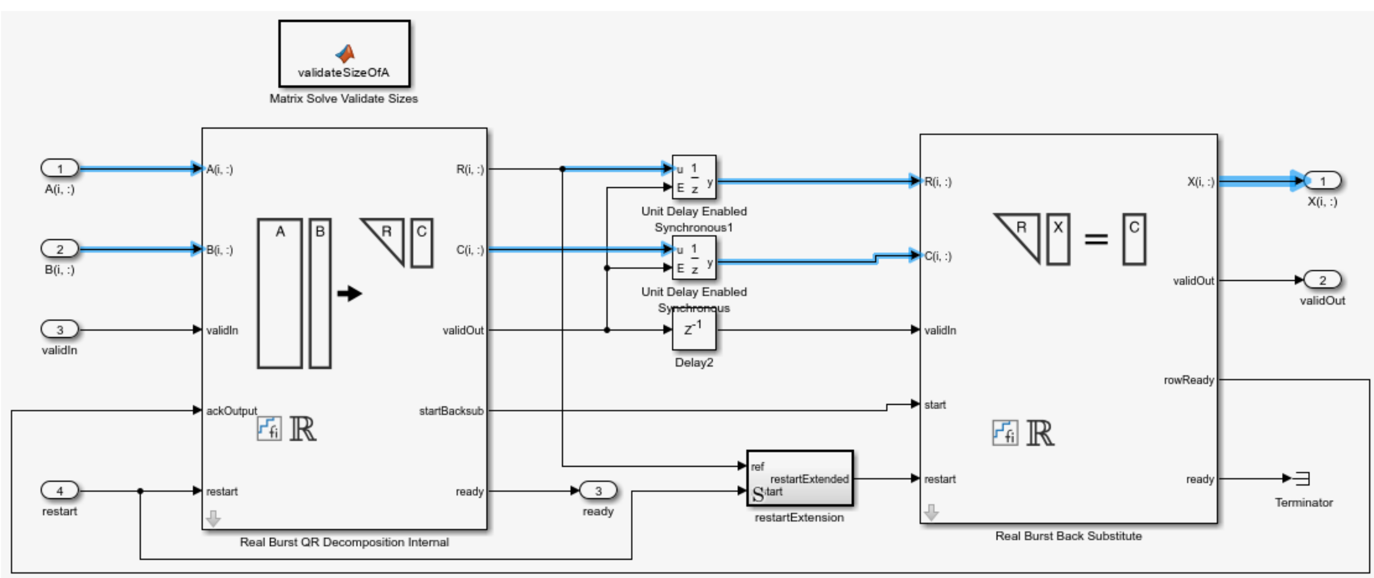
Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

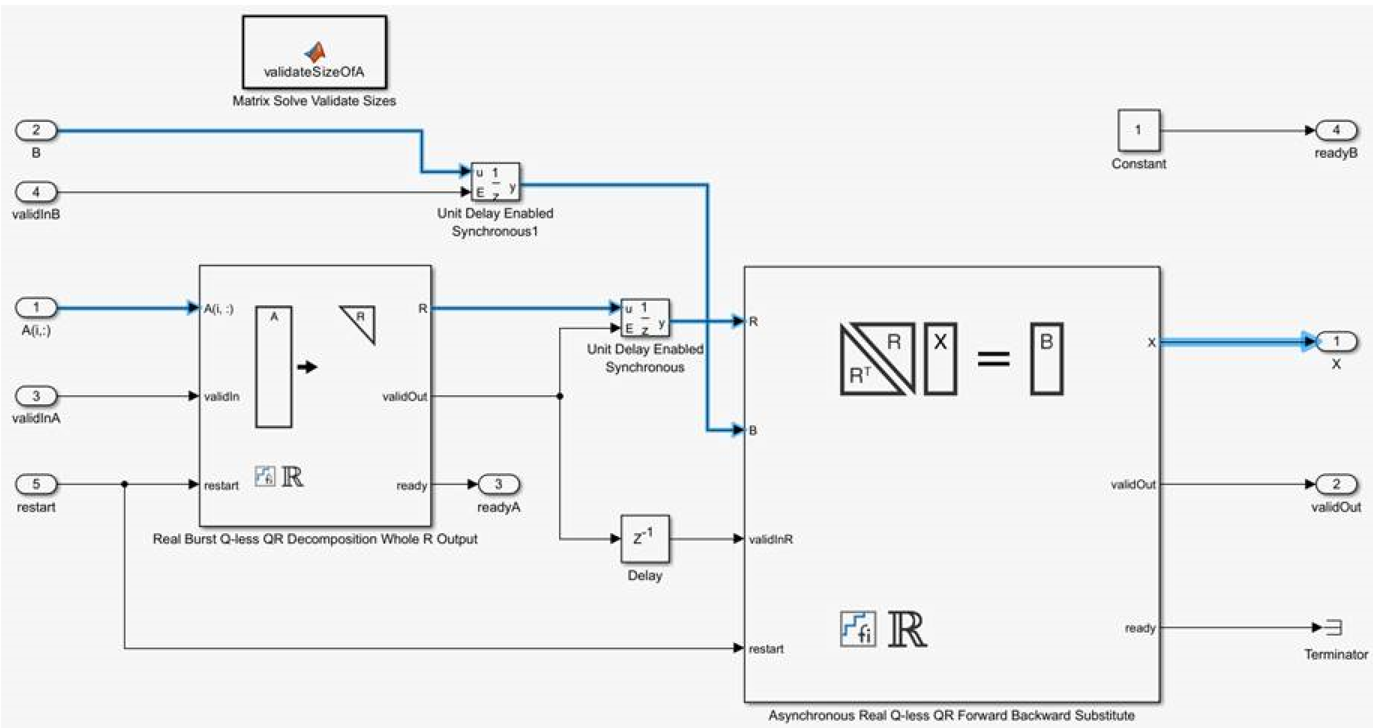
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input A and B matrices into R and C matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input A and B matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input *A* matrix and the resulting *R* matrix is put into a buffer. Then, a forward backward substitution block uses the input *B* matrix and the buffered *R* matrix to compute $R'RX = B$. Because the *R* and *B* matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first *R* and *B* matrices are available. Then it runs continuously using the latest buffered *R* and *B* matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing *A* and *B* matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of *R* and *B* matrices.



The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

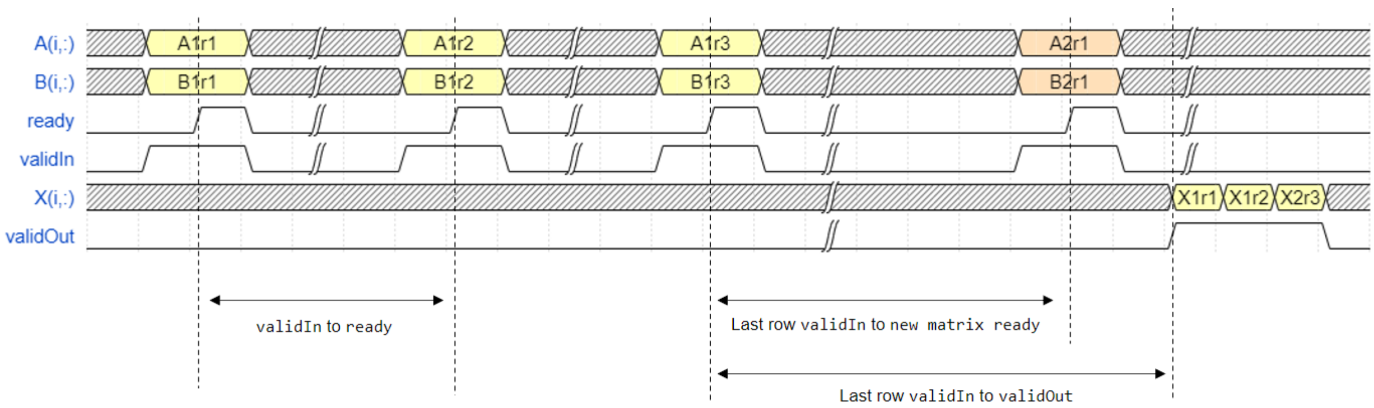
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

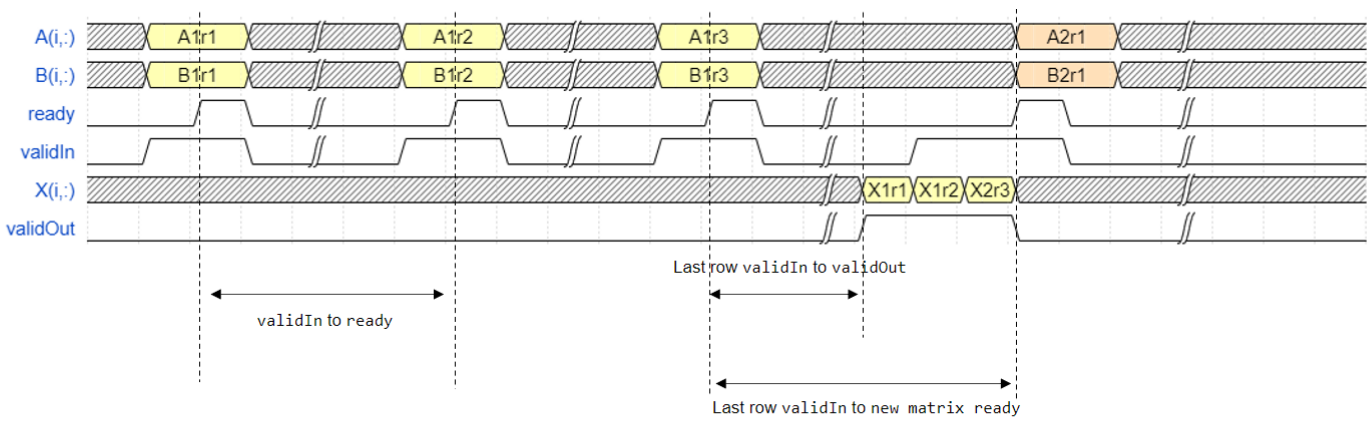


In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Matrix Solve Using Q-less QR Decomposition* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *X* matrix row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `X1r3` is the third row of the first *X* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row within one matrix.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

- Last row `validIn` to new matrix ready — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	**Operation**	**`validIn` to `ready` (cycles)**	**Last Row `validIn` to `validOut` (cycles)**	**Last row `validIn` to new matrix ready (cycles)**
Real Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$(wl + 5)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl + 5)*n + 3.5*(n - 1)^2 + (n - 1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Complex Burst Matrix Solve Using QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$(wl*2 + 11)*n + 3.5*n^2 + n*(\text{nextPow2}(wl) + wl + 8.5) + 3$	$(wl*2 + 11)*n + 3.5*(n-1)^2 + (n-1)(\text{nextPow2}(wl) + wl + 8.5) + 3$
Real Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl + 5)*n + 2$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 27*n + 6 + 3*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$
Complex Burst Matrix Solve Using Q-less QR Decomposition	Synchronous	$(wl*2 + 11)*n + 2$	$7*2^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl)$	$7*n^2 + 33*n + 6 + 4*n*wl + 2*n*\text{nextPow2}(wl) + \min(m,n)$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`

    - Matrix *A* dimension: 16-by-16
    - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	9629	425280	2.26
CLB Registers	10005	850560	1.18
DSPs	2	4272	0.05
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	2.893 ns
Slack	0.421 ns
Clock Frequency	343.37 MHz

Version History

Introduced in R2019b

R2022a: Support for Tikhonov regularization parameter

The Real Burst Matrix Solve Using QR Decomposition block now supports the Tikhonov "Regularization parameter" on page 2-0 .

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

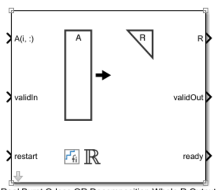
Blocks
Complex Burst Matrix Solve Using QR Decomposition | Real Burst Matrix Solve Using Q-less QR Decomposition | Real Partial-Systolic Matrix Solve Using QR Decomposition

Functions
`fixed.qrAB`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst Q-less QR Decomposition Whole R Output

Q-less QR decomposition for real-valued matrices



Real Burst Q-less QR Decomposition Whole R Output

Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Burst Q-less QR Decomposition Whole R Output block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, where A is a real-valued matrix, without computing Q. The solution to $A'Ax = B$ is $x = R\backslash R'\backslash b$.

When “Regularization parameter” on page 2-0 is nonzero, the Real Burst Q-less QR Decomposition Whole R Output block computes the upper-triangular factor R of the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the regularization parameter.

Ports

Input

A(i,:) — Rows of real matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If A is a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Economy size QR decomposition matrix *R*
vector

Economy size QR decomposition matrix *R*, returned as a vector. *R* is an upper triangular matrix. The size of matrix *R* is `min(m,n)`-by-`n`. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in input matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Regularization parameter — Regularization parameter
`0` (default) | real nonnegative scalar

Regularization parameter, specified as a real nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

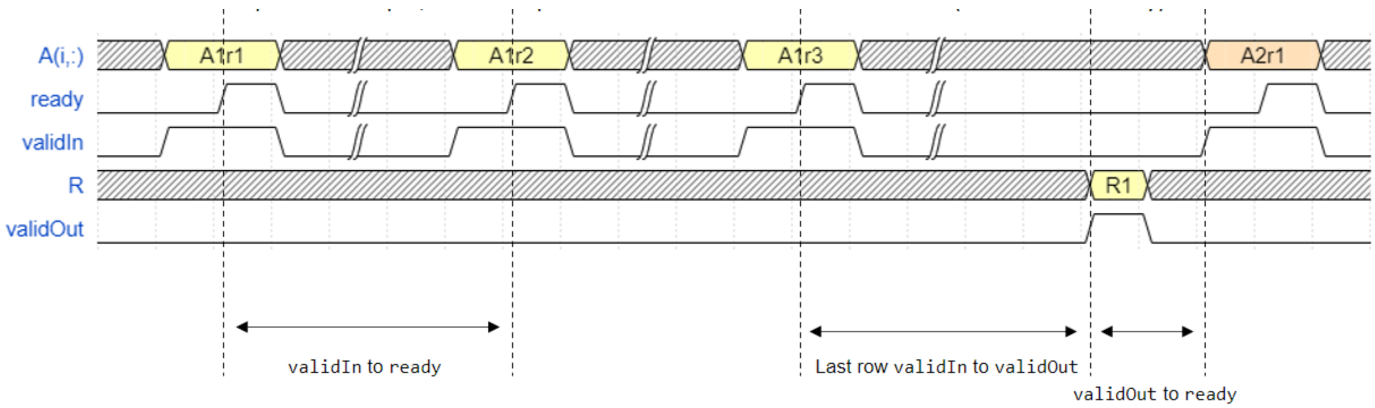
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Q-less QR Decomposition Whole R Output* blocks accept and process *A* and *B* matrices row by row synchronously. After accepting *m* rows, the block outputs the *R* matrix as a single vector.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Q-less QR Decomposition Whole R Output* blocks.

Block	validIn to ready (cycles)	Last Row validIn to validOut (cycles)	validOut to ready (cycles)
Real Burst Q-less QR Decomposition Whole R Output	(*wl* + 5)*min(*m*,*n*) + 2	(*wl* + 5)*min(*m*,*n*) + 2 + min(*m*,*n*) - 1	2
Complex Burst Q-less QR Decomposition Whole R Output	(*wl**2 + 11)*min(*m*,*n*) + 2	(*wl**2 + 11)*min(*m*,*n*) + 2 + min(*m*,*n*) -1	2

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - Matrix *A* dimension: 16-by-16
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	7916	425280	1.86
CLB Registers	8273	850560	0.97
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	2.762 ns
Slack	0.466 ns
Clock Frequency	348.76 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

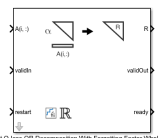
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Complex Burst Q-less QR Decomposition Whole R Output | Real Burst Q-less QR Decomposition | Real Burst QR Decomposition

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output

Q-less QR decomposition for real-valued matrices with infinite number of rows



Real Burst Q-less QR Decomposition With Forgetting Factor Whole R Output

Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition, $A = QR$, without computing Q. A is an infinitely tall real-valued matrix representing streaming data.

When the regularization parameter is nonzero, the Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output block initializes the first upper-triangular factor R to λI_n before factoring in the rows of A, where λ is the regularization parameter and $I_n = \text{eye}(n)$

Ports

Input

A(i,:) — Rows of real matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If A uses a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Economy size QR decomposition matrix *R*
vector

Economy size QR decomposition matrix *R*, returned as a vector. *R* is an upper triangular matrix. The size of matrix *R* is n-by-n. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of columns in matrix A — Number of columns in input matrix *A*
`4` (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Forgetting factor — Forgetting factor applied after each row of the matrix is factored
`0.99` (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of *A* is input indefinitely.

Programmatic Use
Block Parameter: `forgetting_factor`
Type: character vector
Values: real positive scalar
Default: `0`

Regularization parameter — Regularization parameter
`0` (default) | real nonnegative scalar

Regularization parameter, specified as a real nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

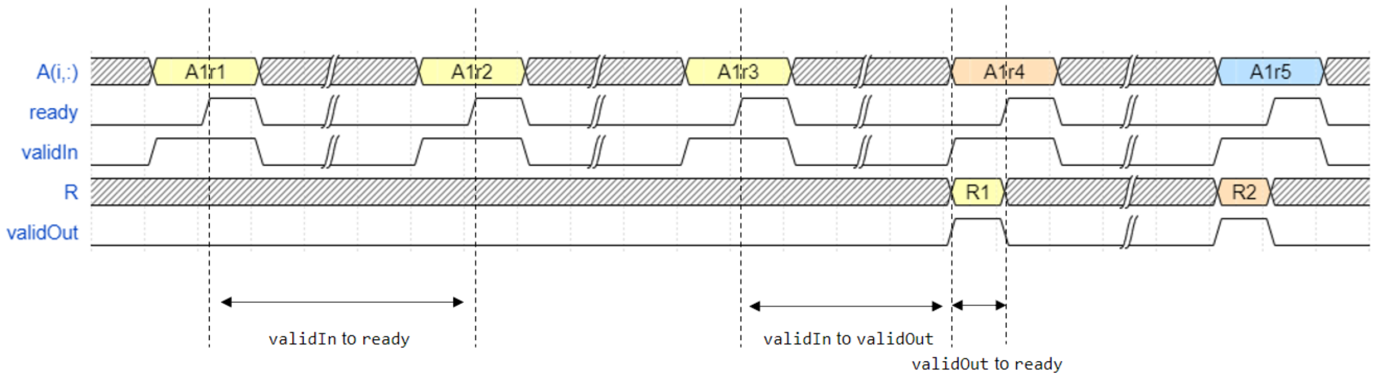
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Q-less QR Decomposition with Forgetting Factor Whole R Output* blocks accept and process the matrix *A* row by row. After accepting the first *m* rows, the block starts to output the *R* matrix as a vector. Then, for each row input, the block calculates an *R* matrix.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- `validIn` to `validOut` — From a successful row input to the block starting to output the corresponding solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst Q-less QR Decomposition with Forgetting Factor Whole R Output* blocks.

Block	`validIn` to `ready` (cycles)	`validIn` to `validOut` (cycles)	`validOut` to `ready` (cycles)
Real Burst Q-less QR Decomposition with Forgetting Factor Whole R Output	(*wl* + 5)**n* + 2 + *n*	(*wl* + 5)**n* + 2 + *n* - 1	1
Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output	(*wl**2 + 11)**n* + 2 + *n*	(*wl**2 + 11)**n* + 2 + *n* - 1	1

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `n = 16`
  - Matrix *A* dimension: inf-by-16
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization (%)
CLB LUTs	8015	425280	1.88
CLB Registers	8289	850560	0.97
DSPs	16	4272	0.37
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	2.872 ns
Slack	0.441 ns
Clock Frequency	345.74 MHz

Version History

Introduced in R2022b

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

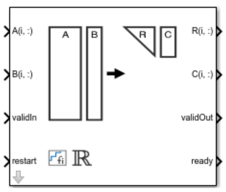
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Complex Burst Q-less QR Decomposition with Forgetting Factor Whole R Output | Real Burst Q-less QR Decomposition Whole R Output

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Burst QR Decomposition

QR decomposition for real-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Burst QR Decomposition block uses QR decomposition to compute R and $C = Q'B$, where $QR = A$, and A and B are real-valued matrices. The least-squares solution to $Ax = B$ is $x = R\backslash C$. R is an upper triangular matrix and Q is an orthogonal matrix. To compute $C = Q'$, set B to be the identity matrix.

When “Regularization parameter” on page 2-0 is nonzero, the Real Burst QR Decomposition block transforms $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ in-place to $R = Q'\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ and $\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ in-place to $C = Q'\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ where λ is the regularization parameter, QR is the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of matrix *B*
vector

Rows of real matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values on the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R(i,:) — Rows of matrix *R*
scalar | vector

Rows of the economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper triangular matrix. The size of the matrix *R* is min(*m*,*n*)-by-*n*. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

C(i,:) — Rows of matrix *C* = *Q*'*B*
scalar | vector

Rows of the economy size QR decomposition matrix *C*=*Q*'*B*, returned as a scalar or vector. *C* has the same number of rows as *R*. *C* has the same data type as *B*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether output data is valid, returned as a Boolean scalar. This control signal indicates when the data at output ports `R(i,:)` and `C(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed the *R* and *C* matrices. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether block is ready, returned as a Boolean scalar. This control signal that indicates when the block is ready for new input data. When this value is 1 (`true`) and the `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in matrices *A* and *B*
4 (default) | positive integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Tips

Use `fixed.getQRDecompositionModel(A,B)` to generate a template model containing a Real Burst QR Decomposition block for real-valued input matrices `A` and `B`.

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

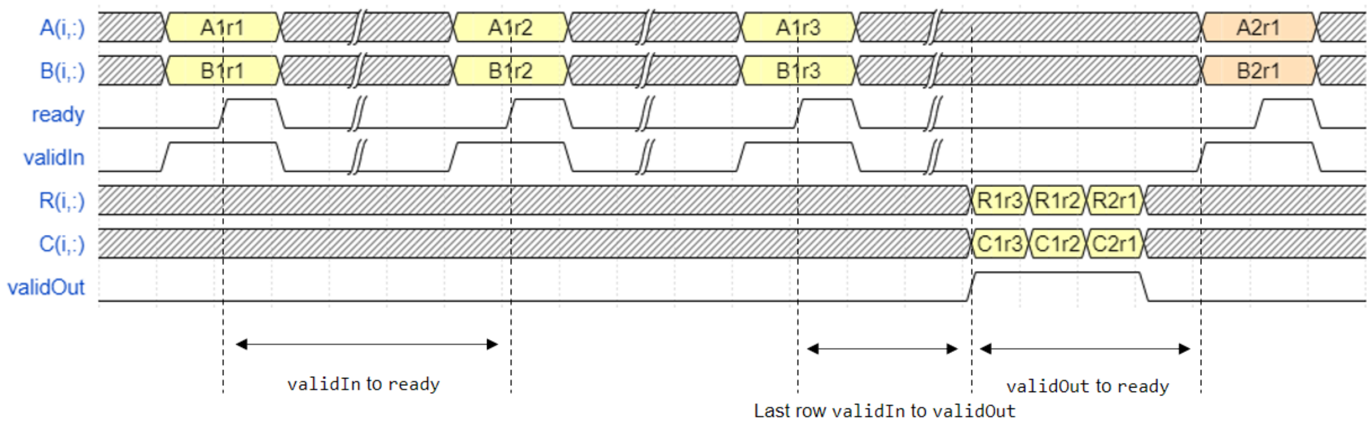
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst QR Decomposition* blocks accept and process A and B matrices row by row synchronously. After accepting m rows, the block outputs the R and C matrices row by row continuously. The matrices are output from the last row to the first row.

For example, assume that the input A and B matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

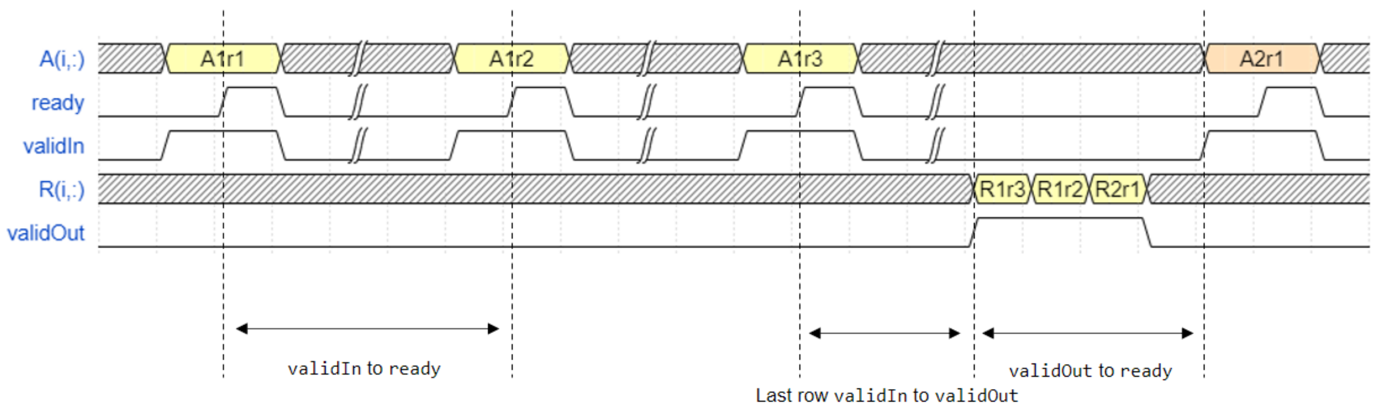


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.
- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The *Burst Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the matrix *R* row by row continuously. The matrix is output from the last row to the first row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1r3` is the third row of the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

- `validOut` to `ready` — From the block starting to output the solution to the block ready to accept the next matrix input.

The following table provides details of the timing for the *Burst QR Decomposition* blocks.

Block	**`validIn` to `ready` (cycles)**	**Last Row `validIn` to `validOut` (cycles)**	**`validOut` to `ready` (cycles)**
Real Burst QR Decomposition	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2$	$\min(m,n) + 1$
Complex Burst QR Decomposition	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2$	$\min(m,n) + 1$
Real Burst Q-less QR Decomposition	$(wl + 5)*\min(m,n) + 2$	$(wl + 5)*\min(m,n) + 2$	$\min(m,n) + 1$
Complex Burst Q-less QR Decomposition	$(wl*2 + 11)*\min(m,n) + 2$	$(wl*2 + 11)*\min(m,n) + 2$	$\min(m,n) + 1$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix A dimension: 16-by-16
  - Matrix B dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	**Usage**	**Available**	**Utilization (%)**
CLB LUTs	8646	425280	2.03

Resource	Usage	Available	Utilization (%)
CLB Registers	8797	850560	1.03
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.104 ns
Slack	0.211 ns
Clock Frequency	320.27 MHz

Version History

Introduced in R2019b

R2022a: Support for Tikhonov regularization parameter

The Real Burst QR Decomposition block now supports the Tikhonov "Regularization parameter" on page 2-0 .

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

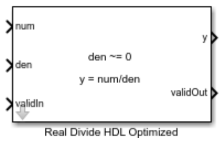
Blocks
Complex Burst QR Decomposition | Real Burst Q-less QR Decomposition | Real Partial-Systolic QR Decomposition

Functions
`fixed.qrAB`

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Real Divide HDL Optimized

Divide one real input by another and generate optimized HDL code



Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Real Divide HDL Optimized block outputs the result of dividing the real scalar `num` by the real scalar `den`, such that `y = num/den`.

Limitations

Data type override is not supported for the Real Divide HDL Optimized block.

Ports

Input

num — Numerator
real scalar

Numerator, specified as a real scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

den — Denominator
real scalar

Denominator, specified as a real scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether input is valid
`Boolean` scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `num` and `den` input ports are valid. When this value is `1` (`true`), the block captures the values at the input ports `num` and `den`. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

y — Output computed by dividing inputs
real scalar

Output computed by dividing `num` by `den`, such that `y = num/den`, returned as a real scalar with the data type specified by the **Output datatype** parameter.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output at port `y`. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Parameters

Output datatype — Data type of the output
`fixdt(1,18,10)` (default) | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output `y`, specified as `fixdt(1,18,10)`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly or expressed as a data type object, such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,10)'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,10)'`

Tips

The blocks Divide by Constant HDL Optimized, Real Divide HDL Optimized, and Complex Divide HDL Optimized all perform the division operation and generate optimized HDL code.

- Real Divide HDL Optimized and Complex Divide HDL Optimized are based on a CORIDC algorithm. These blocks accept a wide variety of inputs, but will result in greater latency.
- Divide by Constant HDL Optimized accepts only real inputs and a constant divisor. Use of this block consumes DSP slices, but will complete the division operation in fewer cycles and at a higher clock rate.

Algorithms

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers.

Fully Pipelined Fixed-Point Computations

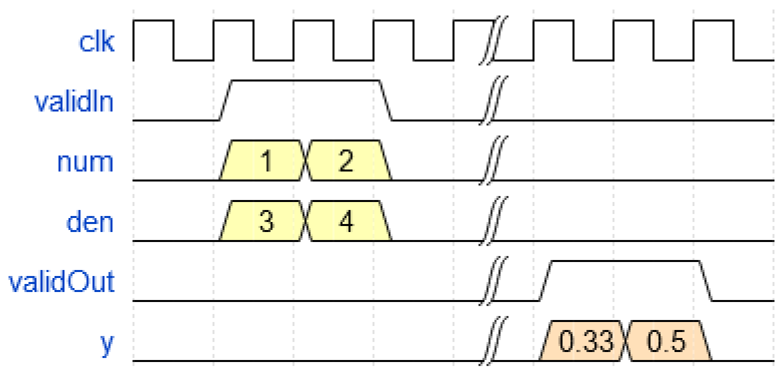
The Real Divide HDL Optimized block supports HDL code generation for fixed-point data with binary-point scaling. It is designed with this application in mind, and employs hardware specific semantics and optimizations. One of these optimizations is pipelining its entire internal circuitry to maintain a very high throughput.

When deploying intricate algorithms to FPGA or ASIC devices, there is often a trade-off between resource usage and total throughput for a given computation. Resource-sharing often reduces the resources consumed by a design, but also reduces the throughput in the process. Simple arithmetic and trigonometric computations, which typically form parts of bigger computations, require high throughput to drive circuits further in the design. Thus, fully pipelined implementations consume more on-chip resources but are beneficial in large designs.

All of the key computational units in the Real Divide HDL Optimized block are fully pipelined internally. This includes not only the CORDIC circuitry used to perform the Givens rotations, but also the adders and shifters used elsewhere in the design, thus ensuring maximum throughput.

How to Interface with the Real Divide HDL Optimized Block

Because of its fully pipelined nature, the Real Divide HDL Optimized block is able to accept input data on any cycle, including consecutive cycles. To send input data to the block, the `validIn` signal must be true. When the block has finished the computation and is ready to send the output, it will change `validOut` to true for one clock cycle. For inputs sent on consecutive cycles, `validOut` will also be set to true on consecutive cycles. Both the numerator and the denominator must be sent together on the same cycle.



The latency depends on the input data type, as summarized in the table. The word length of the inputs `num` and `den` can differ. In the table, `u` represents the input with the larger word length.

Input Type	Latency
`fi`	`1 + (nextpow2(u.WordLength + 1 - issigned(u)) + 1) + 2 + u.WordLength + 2 - issigned(u) + 5`
Scaled double	`1 + 2 + u.WordLength + 2 - issigned(u) + 5`
Floating point	8

Division by Zero Behavior

For fixed-point inputs `num` and `den`, the Real Divide HDL Optimized block saturates on overflow for division by zero. The behavior for fixed-point division by zero is summarized in the table below.

Wrap Overflow	Saturate Overflow
0/0 = 0	0/0 = 0
1/0 = 0	1/0 = upper bound
-1/0 = 0	-1/0 = lower bound

For floating-point inputs, the Real Divide HDL Optimized block follows IEEE Standard 754.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Restrictions

Supports binary-point scaled fixed-point data types only.

See Also

Blocks

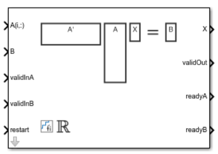
Complex Divide HDL Optimized | Real Reciprocal HDL Optimized | Normalized Reciprocal HDL Optimized

Functions

`fixed.cordicReciprocal` | `fixed.cordicDivide`

Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition

Compute value of X in the equation $A'AX = B$ for real-valued matrices using Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition block solves the system of linear equations $A'AX = B$ using Q-less QR decomposition, where A and B are real-valued matrices.

When “Regularization parameter” on page 2-0 is nonzero, the Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition block solves the matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the regularization parameter, A is an m-by-n matrix, and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of real matrix A
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B — Matrix B
vector | matrix

Real matrix B, specified as a vector or matrix. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether A input is valid
`Boolean` scalar

Whether `A(i, :)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (`true`), the block captures the values at the `A(i,:)` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

validInB — Whether B input is valid
`Boolean` scalar

Whether `B` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `B` input port is valid. When this value is `1` (`true`) and the `readyB` value is `1` (`true`), the block captures the values at the `B` input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validInA` and `validInB` values are 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X — Matrix *X*
vector | matrix

Matrix *X*, returned as a vector or matrix.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X` is valid. When this value is `1` (`true`), the block has successfully computed a row of *X*. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

readyA — Whether block is ready for input A
`Boolean` scalar

Whether the block is ready for input `A(i, :)`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validInA` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input B
`Boolean` scalar

Whether the block is ready for input `B`, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validInB` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
4 (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix B — Number of columns in matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`

Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

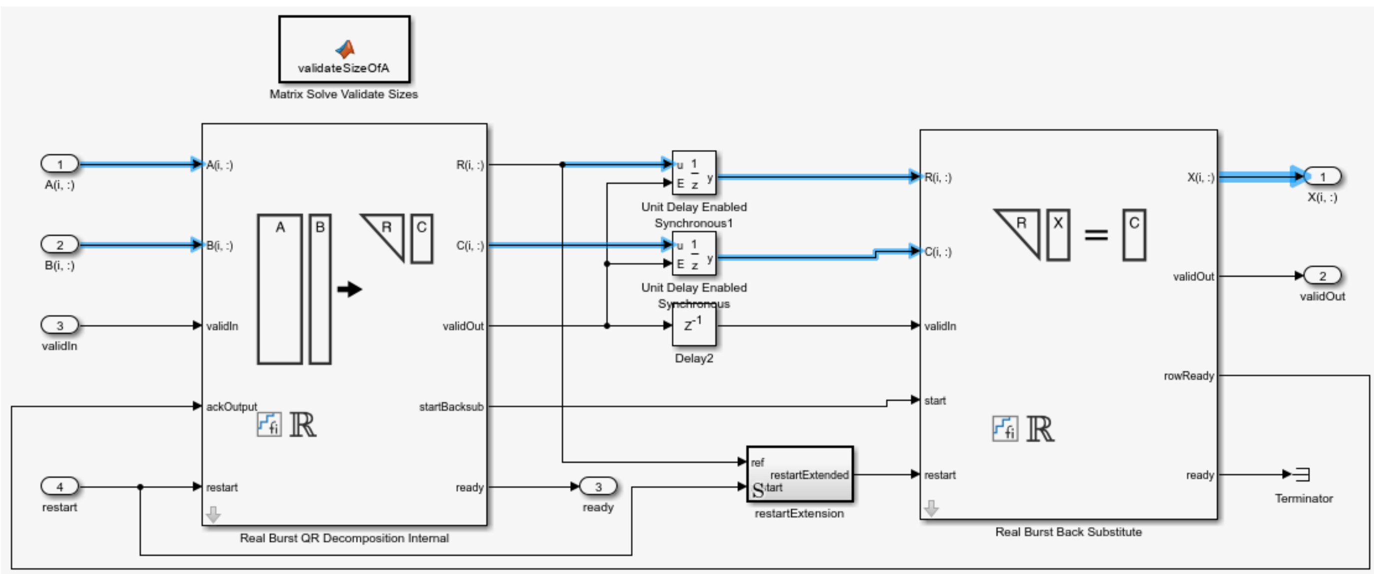
For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

AMBA AXI Handshake Process

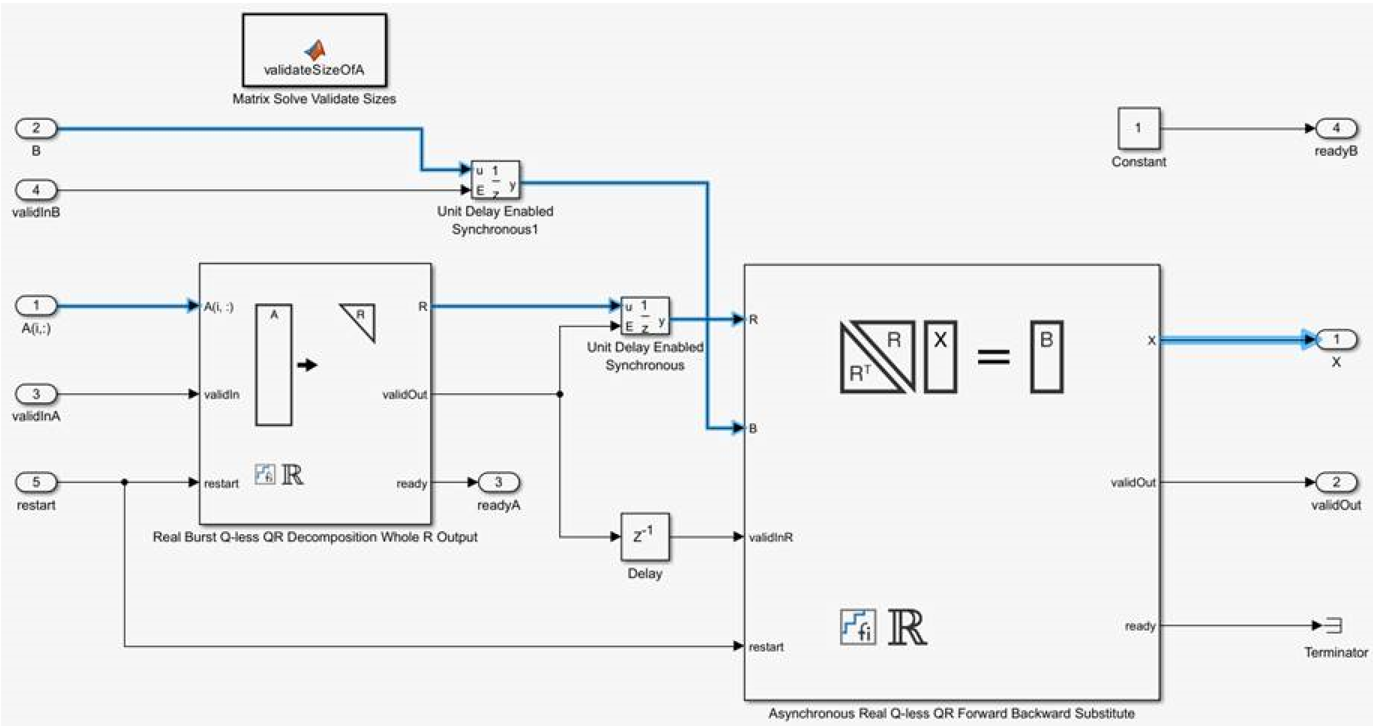
This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Synchronous vs Asynchronous Implementation

The *Matrix Solve Using QR Decomposition* blocks operate synchronously. These blocks first decompose the input *A* and *B* matrices into *R* and *C* matrices using a QR decomposition block. Then, a back substitute block computes $RX = C$. The input *A* and *B* matrices propagate through the system in parallel, in a synchronized way.



The *Matrix Solve Using Q-less QR Decomposition* blocks operate asynchronously. First, Q-less QR decomposition is performed on the input *A* matrix and the resulting *R* matrix is put into a buffer. Then, a forward backward substitution block uses the input *B* matrix and the buffered *R* matrix to compute $R'RX = B$. Because the *R* and *B* matrices are stored separately in buffers, the upstream Q-less QR decomposition block and the downstream Forward Backward Substitute block can run independently. The Forward Backward Substitute block starts processing when the first *R* and *B* matrices are available. Then it runs continuously using the latest buffered *R* and *B* matrices, regardless of the status of the Q-less QR Decomposition block. For example, if the upstream block stops providing *A* and *B* matrices, the Forward Backward Substitute block continues to generate the same output using the last pair of *R* and *B* matrices.

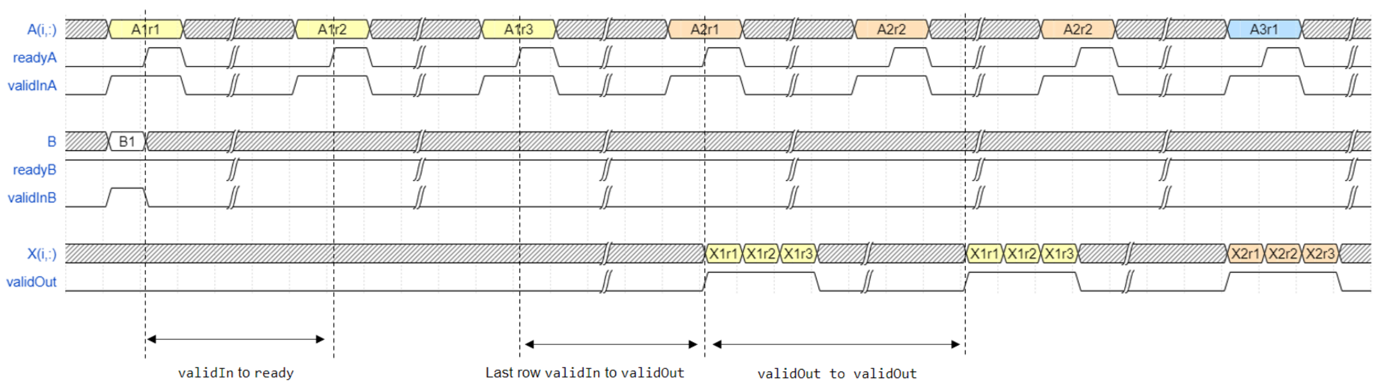


The *Burst (Asynchronous) Matrix Solve Using Q-less QR Decomposition* blocks are available in both synchronous and asynchronous operation variants, as denoted by the block name.

Block Timing

The *Burst Asynchronous Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



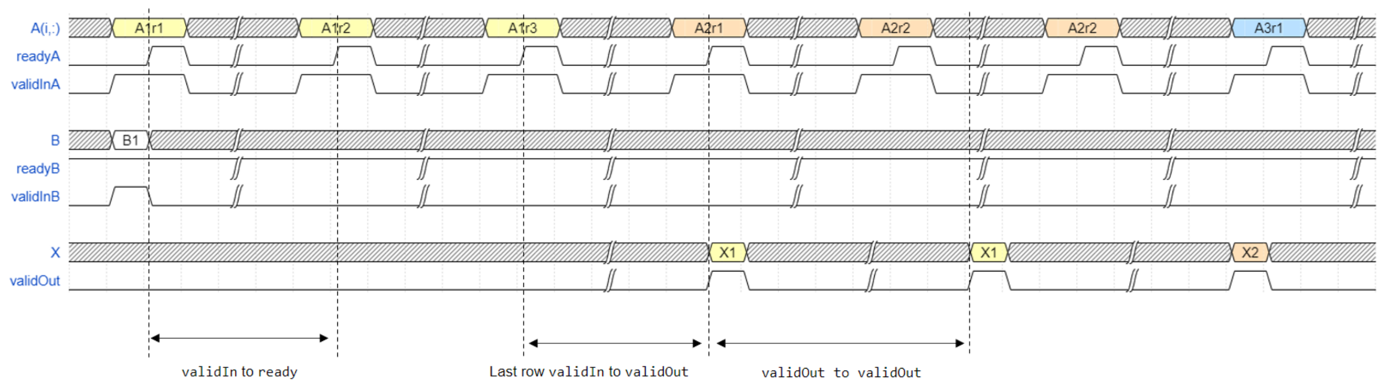
In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.

- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using QR Decomposition* and *Burst Matrix Solve Using Q-less QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	`validOut` to `validOut` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl+5)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Asynchronous Matrix Solve Using Q-less QR Decomposition	Asynchronous	$(wl*2 + 11)*n + 2 + (n + 1)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, m represents the number of rows in matrix A, and n is the number of columns in matrix A. wl represents the word length of A.

- If the data type of A is fixed point, then wl is the word length.
- If the data type of A is double, then wl is 53.
- If the data type of A is single, then wl is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`

  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	104968	425280	24.68
CLB Registers	90547	850560	10.65
DSPs	4	4272	0.09
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.785 ns
Slack	0.197 ns
Clock Frequency	262.95 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

This block depends on a partial-systolic QR decomposition block. Since 23a, when you update the diagram, the loop which composes the partial-systolic pipeline in the QR decomposition block is unrolled. This updated internal architecture removes dead operations in simulation and generated code, thus requiring fewer hardware resources. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

R2022a: Support for Tikhonov regularization parameter

The Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition block now supports the Tikhonov “Regularization parameter” on page 2-0 .

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

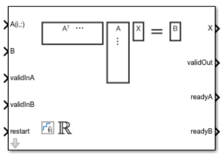
Blocks
Complex Partial-Systolic Q-less QR Decomposition | Real Partial-Systolic Matrix Solve Using QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Real Burst Matrix Solve Using Q-less QR Decomposition

Functions
`fixed.qlessQRMatrixSolve`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Compute value of X in the equation $A'AX = B$ for real-valued matrices with infinite number of rows using Q-less QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block solves the system of linear equations $A'AX = B$ using Q-less QR decomposition, where A and B are real-valued matrices. A is an infinitely tall matrix representing streaming data.

When the regularization parameter is nonzero, the Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor initializes the first upper-triangular factor R to λI_n before factoring in the rows of A, where λ is the regularization parameter and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of real matrix A
vector

Rows of real matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B — Matrix B
matrix

Real matrix B, specified as a matrix. B is an n-by-p matrix where $n \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validInA — Whether A input is valid
`Boolean` scalar

Whether `A(i, :)` input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is `1` (`true`) and the `readyA` value is `1` (true), the block captures the values at the A(i,:) input port. When this value is 0 (false), the block ignores the input samples.

After sending a true validInA signal, there may be some delay before readyA is set to false. To ensure all data is processed, you must wait until readyA is set to false before sending another true validInA signal.

Data Types: Boolean

validInB — Whether B input is valid
Boolean scalar

Whether B input is valid, specified as a Boolean scalar. This control signal indicates when the data from the B input port is valid. When this value is 1 (true) and the readyB value is 1 (true), the block captures the values at the B input port. When this value is 0 (false), the block ignores the input samples.

After sending a true validInB signal, there may be some delay before readyB is set to false. To ensure all data is processed, you must wait until readyB is set to false before sending another true validInB signal.

Data Types: Boolean

restart — Whether to clear internal states
Boolean scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (true), the block stops the current calculation and clears all internal states. When this value is 0 (false) and the validInA and validInB values are both 1 (true), the block begins a new subframe.

Data Types: Boolean

Output

X — Matrix *X*
vector | matrix

Matrix *X*, returned as a vector or matrix.

Data Types: single | double | fixed point

validOut — Whether output data is valid
Boolean scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port X is valid. When this value is 1 (true), the block has successfully computed a row of *X*. When this value is 0 (false), the output data is not valid.

Data Types: Boolean

readyA — Whether block is ready for input A
Boolean scalar

Whether the block is ready for input A, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (true) and validInA value is 1 (true), the block accepts input data in the next time step. When this value is 0 (false), the block ignores input data in the next time step.

After sending a `true` `validInA` signal, there may be some delay before `readyA` is set to `false`. To ensure all data is processed, you must wait until `readyA` is set to `false` before sending another `true` `validInA` signal.

Data Types: `Boolean`

readyB — Whether block is ready for input B
`Boolean` scalar

Whether the block is ready for input B, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and `validInB` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validInB` signal, there may be some delay before `readyB` is set to `false`. To ensure all data is processed, you must wait until `readyB` is set to `false` before sending another `true` `validInB` signal.

Data Types: `Boolean`

Parameters

Number of columns in matrix A and rows in matrix B — Number of columns in matrix *A* and rows in matrix *B*
`4` (default) | positive integer-valued scalar

Number of columns in matrix *A* and rows in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Forgetting factor — Forgetting factor applied after each row of matrix is factored
0.99 (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of *A* is input indefinitely.

Programmatic Use
Block Parameter: `forgettingFactor`
Type: character vector
Values: positive integer-valued scalar

Default: `0.99`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,14)'`

Tips

- Use `fixed.forgettingFactor` to compute the forgetting factor, α, for an infinite number of rows with the equivalent gain of a matrix with m rows.
- Use `fixed.forgettingFactorInverse` to compute the number of rows, m, of a matrix with equivalent gain corresponding to forgetting factor α

  .

Algorithms

Q-less QR Decomposition with Forgetting Factor

The Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block implements the following recursion to compute the upper-triangular factor R of continuously streaming n-by-1 row vectors $A(k,:)$ using forgetting factor α. It's as if matrix A is infinitely tall. The forgetting factor in the range $0 < \alpha < 1$ prevents it from integrating without bound.

$$R_0 = \text{zeros}(n, n)$$
$$[\sim, R_1] = \text{qr}\left(\begin{bmatrix} R_0 \\ A(1, :) \end{bmatrix}, 0\right)$$
$$R_1 = \alpha R_1$$
$$[\sim, R_2] = \text{qr}\left(\begin{bmatrix} R_1 \\ A(2, :) \end{bmatrix}, 0\right)$$
$$R_2 = \alpha R_2$$
$$\vdots$$
$$[\sim, R_k] = \text{qr}\left(\left[\begin{bmatrix} R_{k-1} \\ A(k, :) \end{bmatrix}\right], 0\right)$$
$$R_k = \alpha R_k$$
$$\vdots$$

Q-less QR Decomposition with Forgetting Factor and Tikhonov Regularization

The output X_k after processing the k^{th} input $A(k,:)$ is computed using the following iteration.

$$R_0 = \lambda I_n$$
$$[\sim, R_1] = \text{qr}\left(\begin{bmatrix} R_0 \\ A(1, :) \end{bmatrix}, 0\right)$$
$$R_1 = \alpha R_1$$
$$X_1 = R_1 \backslash (R'_1 \backslash B)$$
$$[\sim, R_2] = \text{qr}\left(\begin{bmatrix} R_1 \\ A(2, :) \end{bmatrix}, 0\right)$$
$$R_2 = \alpha R_2$$
$$X_2 = R_2 \backslash (R'_2 \backslash B)$$
$$\vdots$$
$$[\sim, R_k] = \text{qr}\left(\begin{bmatrix} R_{k-1} \\ A(k, :) \end{bmatrix}, 0\right)$$
$$R_k = \alpha R_k$$
$$X_k = R_k \backslash (R'_k \backslash B)$$
$$\vdots$$

This is mathematically equivalent to computing $A'_k A_k X = B$, where A_k is defined as follows, though the block never actually creates A_k.

$$A_k = \begin{bmatrix} \alpha^k \lambda I_n \\ \begin{bmatrix} \alpha^k & & & \\ & \alpha^{k-1} & & \\ & & \ddots & \\ & & & \alpha \end{bmatrix} A(1:k, :) \end{bmatrix}$$

Forward and Backward Substitution

When an upper triangular factor is ready, then forward and backward substitution are computed with the current input B to produce output X.

$$X = R_k \backslash (R_k' \backslash B)$$

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see "Choose a Block for HDL-Optimized Fixed-Point Matrix Operations".

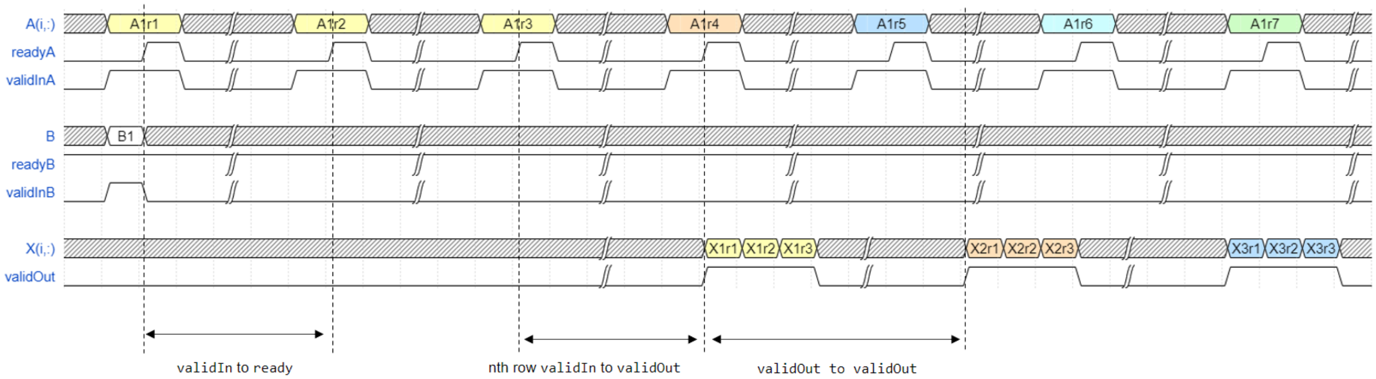
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix A row-by-row and matrix B as a single vector. After accepting the first valid pair of A and B matrices, the block outputs the X matrices row by row continuously. The matrix is output from the first row to the last row.

For example, assume that the input A matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

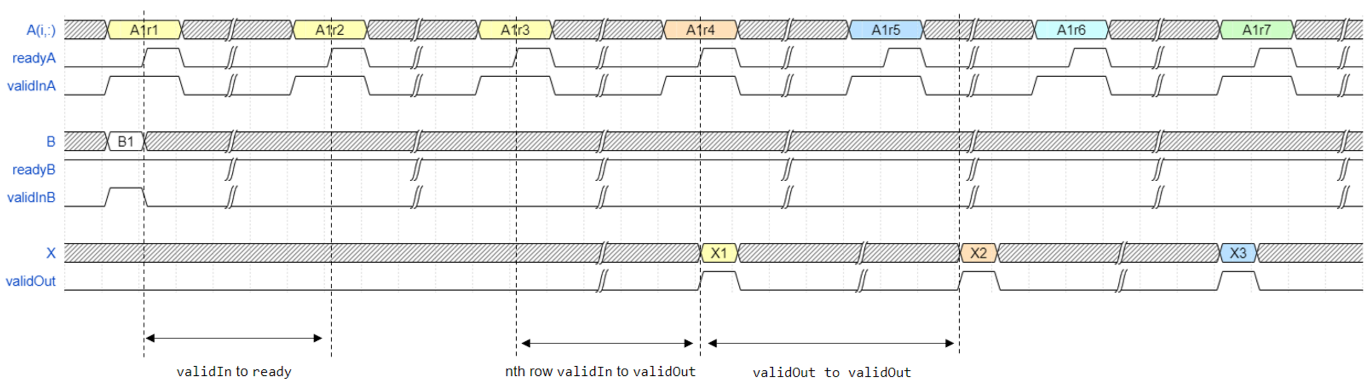


In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.
- n^{th} row `validIn` to `validOut` — From the n^{th} row input to the block starting to output the first solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks accept matrix *A* row-by-row and matrix *B* as a single vector. After accepting the first valid pair of *A* and *B* matrices, the block outputs the *X* matrices row by row continuously.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `A1r2` is the second row of the first *A* matrix, and so on.
- `validIn` to `ready` — From a successful *A* row input to the block being ready to accept the next row.
- `validOut` to `validOut` — Because the Forward Backward Substitution block runs continuously, it generates output at a constant rate. This is the delay between two adjacent valid outputs.

- Last row `validIn` to `validOut` — From the last m^{th} row input to the block starting to output the solution.
- This block is always ready to accept *B* matrices, so `readyB` is always asserted.

The following table provides details of the timing for the *Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* and *Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor* blocks.

Block	**Operation**	**`validIn` to `ready` (cycles)**	**`validOut` to `validOut` (cycles)**	**n^{th} Row `validIn` to `validOut` (cycles)**
Real Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl + 5)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 5)*n + n$
Complex Burst Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$(wl*2 + 11)*n + 2 + n$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n+ 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl*2 + 11)*n + n$
Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 7$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 6)*n + 2$
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor	Asynchronous	$wl + 9$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl)$	$4*n^2 + 25*n + 5 + 2*n*wl + 2*n*\text{nextpow2}(wl) + (wl + 7.5)*2*n + 2$

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: inf-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 250 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	120582	425280	28.35
CLB Registers	90769	850560	10.67
DSPs	4	4272	0.09
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	4 ns
Data Path Delay	3.853 ns
Slack	0.129 ns
Clock Frequency	258.33 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

This block depends on a partial-systolic QR decomposition block. Since 23a, when you update the diagram, the loop which composes the partial-systolic pipeline in the QR decomposition block is unrolled. This updated internal architecture removes dead operations in simulation and generated code, thus requiring fewer hardware resources. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

R2022a: Support for Tikhonov regularization parameter

The Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor block now supports the Tikhonov “Regularization parameter” on page 2-0 .

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

Blocks

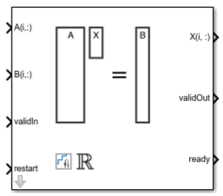
Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic Matrix Solve Using QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Real Burst Matrix Solve Using QR Decomposition

Functions
`fixed.qlessQRMatrixSolve`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Partial-Systolic Matrix Solve Using QR Decomposition

Compute value of x in the equation $Ax = B$ for real-valued matrices using QR decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Linear System Solvers

Description

The Real Partial-Systolic Matrix Solve Using QR Decomposition block solves the system of linear equations $Ax = B$ using QR decomposition, where A and B are real-valued matrices. To compute $x = A^{-1}$, set B to be the identity matrix.

When "Regularization parameter" on page 2-0 is nonzero, the Real Partial-Systolic Matrix Solve Using QR Decomposition block computes the matrix solution of real-valued $\begin{bmatrix}\lambda I_n\\ A\end{bmatrix}X = \begin{bmatrix}0_{n,p}\\ B\end{bmatrix}$ where λ is the regularization parameter, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of real matrix A
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $m \geq n$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of real matrix B
vector

Rows of real matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values on the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

X(i,:) — Rows of matrix *X*
scalar | vector

Rows of the matrix *X*, returned as a scalar or vector.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output port `X(i,:)` is valid. When this value is 1 (`true`), the block has successfully computed a row of matrix *X*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is 1 (`true`) and the `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrices A and B — Number of rows in matrices *A* and *B*
4 (default) | positive integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in matrix *A*
`4` (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in matrix *B*
`1` (default) | positive integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
`0` (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Output datatype — Data type of output matrix *X*
`fixdt(1,18,14)` (default) | `double` | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output matrix *X*, specified as `fixdt(1,18,14)`, `double`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,14)'` | `'double'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`

Default: `'fixdt(1,18,14)'`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see "Choose a Block for HDL-Optimized Fixed-Point Matrix Operations".

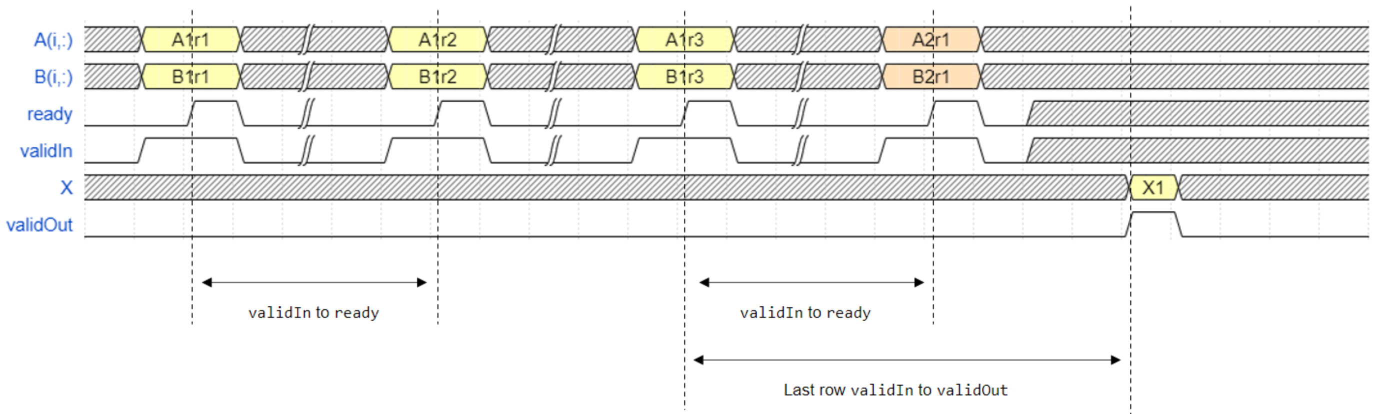
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic Matrix Solve Using QR Decomposition* blocks accept and process A and B matrices row by row. After accepting m rows, the block outputs the matrix X as a single vector. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input A and B matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix and `X1` is the matrix *X*, output as a vector.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The following table provides details of the timing for the *Partial-Systolic Matrix Solve Using QR Decomposition* blocks.

Block	Operation	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Partial-Systolic Matrix Solve Using QR Decomposition	Synchronous	max((wl+7), ceil((3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 1)/n))	(wl + 6)*n + 3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 9 - n
Complex Partial-Systolic Matrix Solve Using QR Decomposition	Synchronous	max((wl + 9), ceil((3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 1)/n))	(wl + 7.5)*2*n + 3.5*n^2 + n*(nextpow2(wl) + wl + 9.5) + 9 - n

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	110589	425280	26.00
CLB Registers	87850	850560	10.33
DSPs	2	4272	0.05
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.163 ns
Slack	0.151 ns
Clock Frequency	314.23 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

This block depends on a partial-systolic QR decomposition block. Since 23a, when you update the diagram, the loop which composes the partial-systolic pipeline in the QR decomposition block is unrolled. This updated internal architecture removes dead operations in simulation and generated code, thus requiring fewer hardware resources. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

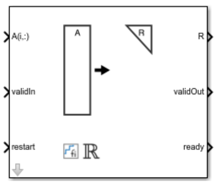
Blocks
Complex Partial-Systolic Matrix Solve Using QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Real Burst Matrix Solve Using QR Decomposition

Functions
`fixed.qrMatrixSolve`

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Real Partial-Systolic Q-less QR Decomposition

Q-less QR decomposition for real-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Partial-Systolic Q-less QR Decomposition block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, where A is a real-valued matrix, without computing Q. The solution to $A'Ax = B$ is $x = R\backslash R'\backslash b$.

When “Regularization parameter” on page 2-0 is nonzero, the Real Partial-Systolic Q-less QR Decomposition block computes the upper-triangular factor R of the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the regularization parameter.

Ports

Input

A(i,:) — Rows of real matrix A
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If A is a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Upper-triangular matrix *R*
matrix

Economy size QR decomposition matrix *R*, returned as a vector or matrix. *R* is an upper triangular matrix. The size of matrix *R* is *n*-by-*n*. The output at `R` has the same data type as the input at `A(i,:)`.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in matrix A — Number of rows in input matrix *A*
4 (default) | positive integer-valued scalar

Number of rows in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	O(n)	O(mn^2)
Partial-Systolic	C	O(m)	O(n^2)
Partial-Systolic with Forgetting Factor	C	O(n)	O(n^2)
Burst	O(n)	O(mn^2)	O(n)

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see "Choose a Block for HDL-Optimized Fixed-Point Matrix Operations".

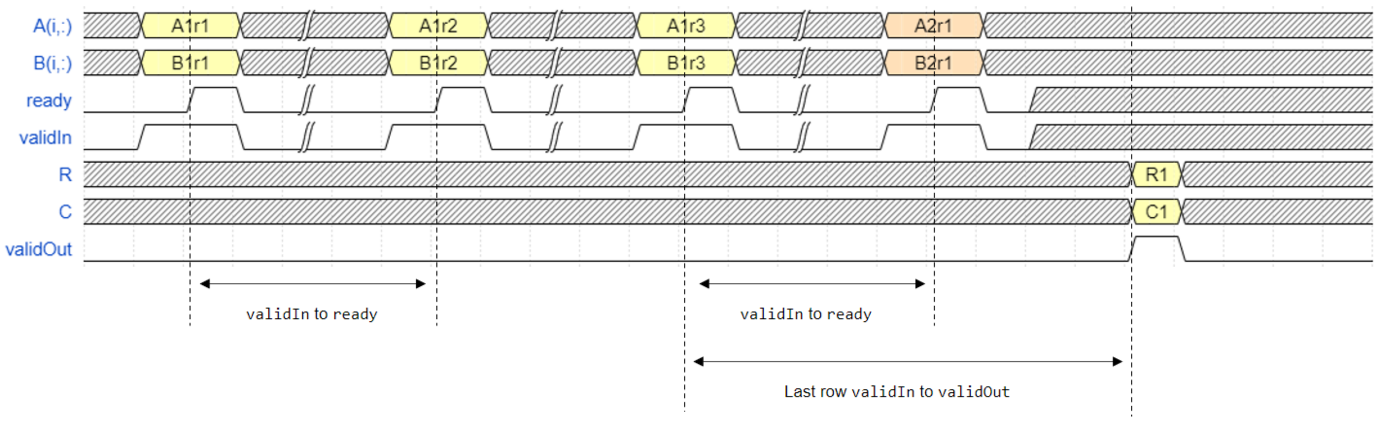
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic QR Decomposition* blocks accept and process A and B matrices row by row. After accepting m rows, the block outputs the R and C matrices as vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input *A* and *B* matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

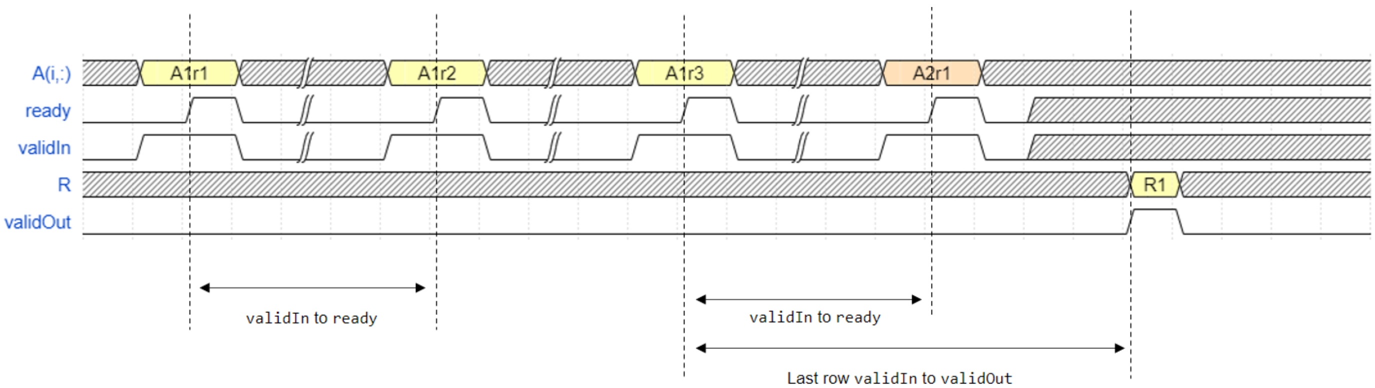


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The *Partial-Systolic Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the *R* matrices as single vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.

- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The following table provides details of the timing for the *Partial-Systolic QR Decomposition* blocks.

Block	`validIn` to `ready` (cycles)	Last Row `validIn` to `validOut` (cycles)
Real Partial-Systolic QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 6
Complex Partial-Systolic QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 6
Real Partial-Systolic Q-less QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 3
Complex Partial-Systolic Q-less QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 3

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The table below shows a summary of the resource utilization results.

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

Resource	Usage
LUT	29896
LUTRAM	994

Resource	Usage
Flip Flop	18953

The tables below show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	96911	425280	22.79
CLB Registers	77355	850560	9.09
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.221 ns
Slack	0.095 ns
Clock Frequency	308.80 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

When you update the diagram, the loop which composes the partial-systolic pipeline is unrolled. This updated internal architecture removes dead operations in simulation and generated code, resulting in a significant decrease in the number of hardware resources required. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

Resource	R2022b	R2023a
LUT	54305	29896
LUTRAM	1090	994
Flip Flop	33901	18953

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

R2022a: Support for Tikhonov regularization parameter

The RealPartial-Systolic Q-less QR Decomposition block now supports the Tikhonov "Regularization parameter" on page 2-0.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

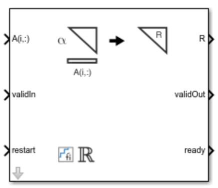
Blocks
Complex Partial-Systolic Q-less QR Decomposition | Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic QR Decomposition | Real Burst Q-less QR Decomposition

Functions
`fixed.qlessQR`

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor

Q-less QR decomposition for real-valued matrices with infinite number of rows



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor block uses QR decomposition to compute the economy size upper-triangular R factor of the QR decomposition $A = QR$, without computing Q. A is an infinitely tall real-valued matrix representing streaming data.

When the regularization parameter is nonzero, the Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor block initializes the first upper-triangular factor R to λI_n before factoring in the rows of A, where λ is the regularization parameter and I_n = `eye(n)`.

Ports

Input

A(i,:) — Rows of real matrix A
vector

Rows of real matrix A, specified as a vector. A is an infinitely tall matrix of streaming data. If A uses a fixed-point data type, A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` input port is valid. When this value is 1 (`true`) and the value of `ready` is 1 (`true`), the block captures the values at the `A(i,:)` input port. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the value at `validIn` is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Upper-triangular matrix *R*
matrix

Economy size QR decomposition matrix *R* multiplied by the `Forgetting factor` parameter, returned as a matrix. *R* is an upper triangular matrix. The size of matrix *R* is *n*-by-*n*. The output at `R` has the same data type as the input at `A(i,:)`.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, specified as a Boolean scalar. This control signal indicates when the data at output port `R` is valid. When this value is 1 (`true`), the block has successfully computed the matrix *R*. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and `validIn` is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: 4

Forgetting factor — Forgetting factor applied after each row of the matrix is factored
0.99 (default) | real positive scalar

Forgetting factor applied after each row of the matrix is factored, specified as a real positive scalar. The output is updated as each row of *A* is input indefinitely.

Programmatic Use
Block Parameter: `forgetting_factor`
Type: character vector
Values: positive integer-valued scalar
Default: `0.99`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Q-less QR Decomposition with Forgetting Factor

The Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor block implements the following recursion to compute the upper-triangular factor *R* of continuously streaming *n*-by-1 row vectors *A(k,:)* using forgetting factor α. It's as if matrix *A* is infinitely tall. The forgetting factor in the range $0 < \alpha < 1$ prevents it from integrating without bound.

$$\begin{aligned}
R_0 &= \text{zeros}(n,n)\\
[\sim, R_1] &= \text{qr}\left(\begin{bmatrix} R_0 \\ A(1,:)\end{bmatrix}, 0\right)\\
R_1 &= \alpha R_1\\
[\sim, R_2] &= \text{qr}\left(\begin{bmatrix} R_1 \\ A(2,:)\end{bmatrix}, 0\right)\\
R_2 &= \alpha R_2\\
&\vdots\\
[\sim, R_k] &= \text{qr}\left(\left[\begin{bmatrix} R_{k-1} \\ A(k,:)\end{bmatrix}\right], 0\right)\\
R_k &= \alpha R_k\\
&\vdots
\end{aligned}$$

Q-less QR Decomposition with Forgetting Factor and Tikhonov Regularization

The upper-triangular factor R_k after processing the k^{th} input $A(k,:)$ is computed using the following iteration.

$$
\begin{aligned}
R_0 &= \lambda I_n \\
[\sim, R_1] &= \mathrm{qr}\left(\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix}, 0\right) \\
R_1 &= \alpha R_1 \\
[\sim, R_2] &= \mathrm{qr}\left(\begin{bmatrix} R_1 \\ A(2,:) \end{bmatrix}, 0\right) \\
R_2 &= \alpha R_2 \\
&\vdots \\
[\sim, R_k] &= \mathrm{qr}\left(\begin{bmatrix} R_{k-1} \\ A(k,:) \end{bmatrix}, 0\right) \\
R_k &= \alpha R_k \\
&\vdots
\end{aligned}
$$

This is mathematically equivalent to computing the upper-triangular factor R_k of matrix A_k, defined as follows, though the block never actually creates A_k.

$$
A_k = \begin{bmatrix} \alpha^k \lambda I_n \\ \begin{bmatrix} \alpha^k & & & \\ & \alpha^{k-1} & & \\ & & \ddots & \\ & & & \alpha \end{bmatrix} A(1:k,:) \end{bmatrix}
$$

Forward and Backward Substitution

When an upper triangular factor is ready, then forward and backward substitution are computed with the current input B to produce output X.

$$
X = R_k \backslash (R_k' \backslash B)
$$

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

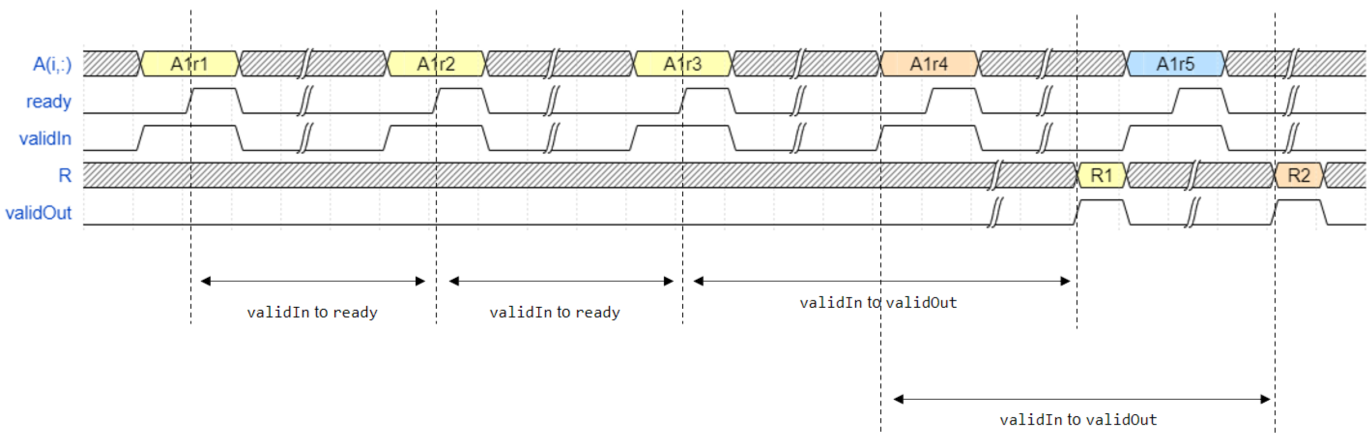
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic QR Decomposition with Forgetting Factor* blocks accept and process the matrix *A* row by row. After accepting the first *m* rows, the block starts to output the *R* matrix as a single vector. From this point, for each row input, the block calculates a *R* matrix. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input matrix *A* is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the Q-less QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- `validIn` to `validOut` — From a successful row input to the block starting to output the corresponding solution.

The following table provides details of the timing for the *Partial-Systolic Q-less QR Decomposition with Forgetting Factor* blocks.

Block	`validIn` to `ready` (cycles)	`validIn` to `validOut` (cycles)
Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor	*wl* + 7	(*wl* + 6)**n* + 3

Block	`validIn` **to** `ready` **(cycles)**	`validIn` **to** `validOut` **(cycles)**
Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor	*wl* + 9	(*wl* + 7.5)*2**n* + 3

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The table below shows a summary of the resource utilization results.

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

Resource	**Usage**
LUT	30190
LUTRAM	10
Flip Flop	17570
BRAM	31

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:

    - `n = 16`
    - `p = 1`
    - Matrix *A* dimension: inf-by-16
    - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	112218	425280	26.39
CLB Registers	77563	850560	9.12
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.191 ns
Slack	0.125 ns
Clock Frequency	311.69 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

When you update the diagram, the loop which composes the partial-systolic pipeline is unrolled. This updated internal architecture removes dead operations in simulation and generated code, resulting in a significant decrease in the number of hardware resources required. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

Resource	R2022b	R2023a
LUT	55482	30190
LUTRAM	10	10
Flip Flop	32375	17570
BRAM	45	31

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:

  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

R2022a: Support for Tikhonov regularization parameter

The Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor block now supports the Tikhonov "Regularization parameter" on page 2-0.

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).

General	
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

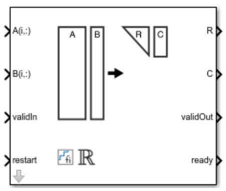
Blocks
Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic Q-less QR Decomposition | Real Partial-Systolic QR Decomposition | Real Burst Q-less QR Decomposition

Functions
`fixed.qlessQR`

Topics
“Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”

Real Partial-Systolic QR Decomposition

QR decomposition for real-valued matrices



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

The Real Partial-Systolic QR Decomposition block uses QR decomposition to compute R and $C = Q'B$, where $QR = A$, and A and B are real-valued matrices. The least-squares solution to $Ax = B$ is $x = R\backslash C$. R is an upper triangular matrix and Q is an orthogonal matrix. To compute $C = Q'$, set B to be the identity matrix.

When “Regularization parameter” on page 2-0 is nonzero, the Real Partial-Systolic QR Decomposition block transforms $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ in-place to $R = Q'\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ and $\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ in-place to $C = Q'\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ where λ is the regularization parameter, QR is the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`.

Ports

Input

A(i,:) — Rows of matrix *A*
vector

Rows of real matrix A, specified as a vector. A is an m-by-n matrix where $m \geq 2$ and $n \geq 2$. If B is single or double, A must be the same data type as B. If A is a fixed-point data type, A must be signed, use binary-point scaling, and have the same word length as B. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

B(i,:) — Rows of matrix *B*
vector

Rows of real matrix B, specified as a vector. B is an m-by-p matrix where $m \geq 2$. If A is single or double, B must be the same data type as A. If B is a fixed-point data type, B must be signed, use binary-point scaling, and have the same word length as A. Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether inputs are valid
`Boolean` scalar

Whether inputs are valid, specified as a Boolean scalar. This control signal indicates when the data from the `A(i,:)` and `B(i,:)` input ports are valid. When this value is 1 (`true`) and the value at `ready` is 1 (`true`), the block captures the values on the `A(i,:)` and `B(i,:)` input ports. When this value is 0 (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

restart — Whether to clear internal states
`Boolean` scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

R — Matrix *R*
scalar | vector

Economy size QR decomposition matrix *R*, returned as a scalar or vector. *R* is an upper triangular matrix. The size of matrix *R* is *n*-by-*n*. *R* has the same data type as *A*.

Data Types: `single` | `double` | `fixed point`

C — Matrix *C* = *Q*'*B*
scalar | vector

Economy size QR decomposition matrix *C*=*Q*'*B*, returned as a scalar or vector. *C* has the same number of rows as *R*. *C* has the same data type as *B*.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether output data is valid, returned as a Boolean scalar. This control signal indicates when the data at output ports `R` and `C` is valid. When this value is 1 (`true`), the block has successfully computed the *R* and *C* matrices. When this value is 0 (`false`), the output data is not valid.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether block is ready, returned as a Boolean scalar. This control signal that indicates when the block is ready for new input data. When this value is 1 (`true`) and the `validIn` value is 1 (`true`), the block accepts input data in the next time step. When this value is 0 (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before `ready` is set to `false`. To ensure all data is processed, you must wait until `ready` is set to `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Number of rows in input matrices A and B — Number of rows in matrices *A* and *B*
4 (default) | positive integer-valued scalar

Number of rows in input matrices *A* and *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `m`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix A — Number of columns in input matrix *A*
4 (default) | positive integer-valued scalar

Number of columns in input matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: character vector
Values: positive integer-valued scalar
Default: `4`

Number of columns in matrix B — Number of columns in input matrix *B*
1 (default) | positive integer-valued scalar

Number of columns in input matrix *B*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `p`
Type: character vector
Values: positive integer-valued scalar
Default: `1`

Regularization parameter — Regularization parameter
0 (default) | real nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Programmatic Use
Block Parameter: `regularizationParameter`
Type: character vector
Values: real nonnegative scalar
Default: `0`

Algorithms

Choosing the Implementation Method

Partial-systolic implementations prioritize speed of computations over space constraints, while burst implementations prioritize space constraints at the expense of speed of the operations. The following table illustrates the tradeoffs between the implementations available for matrix decompositions and solving systems of linear equations.

Implementation	Ready	Latency	Area
Systolic	C	$O(n)$	$O(mn^2)$
Partial-Systolic	C	$O(m)$	$O(n^2)$
Partial-Systolic with Forgetting Factor	C	$O(n)$	$O(n^2)$
Burst	$O(n)$	$O(mn^2)$	$O(n)$

Where C is a constant proportional to the word length of the data, m is the number of rows in matrix A, and n is the number of columns in matrix A.

For additional considerations in selecting a block for your application, see “Choose a Block for HDL-Optimized Fixed-Point Matrix Operations”.

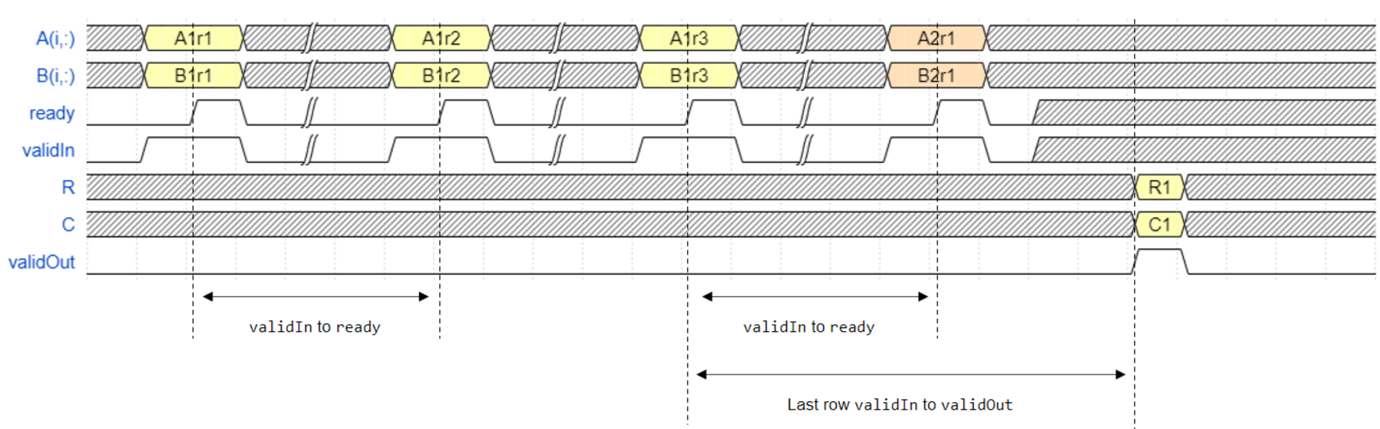
AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate. A `valid` signal indicates when data is available. The `ready` signal indicates that the block can accept the data. Transfer of data occurs only when both the `valid` and `ready` signals are high.

Block Timing

The *Partial-Systolic QR Decomposition* blocks accept and process A and B matrices row by row. After accepting m rows, the block outputs the R and C matrices as vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input A and B matrices are 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.

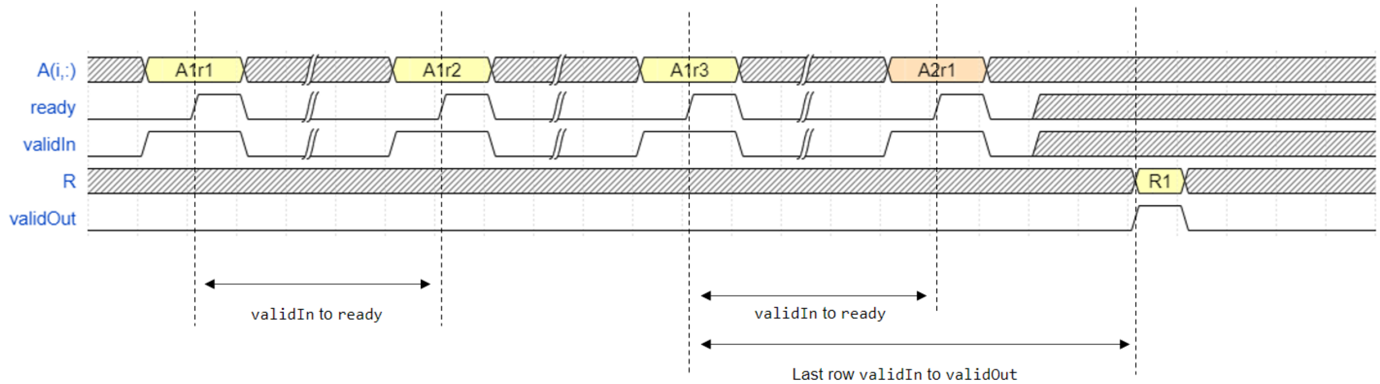


In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The *Partial-Systolic Q-less QR Decomposition* blocks accept and process the matrix *A* row by row. After accepting *m* rows, the block outputs the *R* matrices as single vectors. The partial-systolic implementation uses a pipelined structure, so the block can accept new matrix inputs before outputting the result of the current matrix.

For example, assume that the input *A* matrix is 3-by-3. Additionally assume that `validIn` asserts before `ready`, meaning that the upstream data source is faster than the QR decomposition.



In the figure,

- `A1r1` is the first row of the first *A* matrix, `R1` is the first *R* matrix, and so on.
- `validIn` to `ready` — From a successful row input to the block being ready to accept the next row.
- Last row `validIn` to `validOut` — From the last row input to the block starting to output the solution.

The following table provides details of the timing for the *Partial-Systolic QR Decomposition* blocks.

Block	validIn to ready (cycles)	Last Row validIn to validOut (cycles)
Real Partial-Systolic QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 6
Complex Partial-Systolic QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 6
Real Partial-Systolic Q-less QR Decomposition	*wl* + 7	(*wl* + 6)**n* + 3
Complex Partial-Systolic Q-less QR Decomposition	*wl* + 9	(*wl* + 7.5)*2**n* + 3

In the table, *m* represents the number of rows in matrix *A*, and *n* is the number of columns in matrix *A*. *wl* represents the word length of *A*.

- If the data type of *A* is fixed point, then *wl* is the word length.
- If the data type of *A* is double, then *wl* is 53.
- If the data type of *A* is single, then *wl* is 24.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see “HDL Code Generation and FPGA Synthesis from Simulink Model” (HDL Coder) and “Implement Digital Downconverter for FPGA” (DSP HDL Toolbox).

The table below shows a summary of the resource utilization results.

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

Resource	Usage
LUT	35455
LUTRAM	1126
Flip Flop	22334

The tables below show the post place-and-route resource utilization results and timing summary, respectively.

This example data was generated by synthesizing the block on a Xilinx Zynq UltraScale + RFSoC ZCU111 evaluation board. The synthesis tool was Vivado v.2020.2 (win64).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 16`
  - `n = 16`
  - `p = 1`
  - Matrix *A* dimension: 16-by-16
  - Matrix *B* dimension: 16-by-1
- Input data type: `sfix16_En14`
- Target frequency: 300 MHz

Resource	Usage	Available	Utilization (%)
CLB LUTs	105922	425280	24.91
CLB Registers	82211	850560	9.67
DSPs	0	4272	0.00
Block RAM Tile	0	1080	0.00
URAM	0	80	0.00

	Value
Requirement	3.3333 ns
Data Path Delay	3.276 ns
Slack	0.038 ns
Clock Frequency	303.46 MHz

Version History

Introduced in R2020b

R2023a: Smart unrolling for improved resource utilization

When you update the diagram, the loop which composes the partial-systolic pipeline is unrolled. This updated internal architecture removes dead operations in simulation and generated code, resulting in a significant decrease in the number of hardware resources required. This block simulates with clock and bit-true fidelity with respect to library versions of these blocks in previous releases.

Resource	R2022b	R2023a
LUT	58179	35455
LUTRAM	1330	1126
Flip Flop	37355	22334

This example data was generated by synthesizing the block on a Xilinx Zynq-7 ZC706 evaluation board (-2 speed grade).

The following parameters were used for synthesis.

- Block parameters:
  - `m = 10`
  - `n = 10`
  - `p = 1`
  - Matrix *A* dimension: 10-by-10
  - Matrix *B* dimension: 10-by-1
- Input data type: `sfix18_En12`

References

[1] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only.

See Also

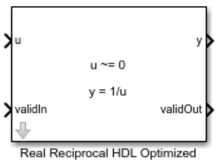
Blocks
Complex Partial-Systolic QR Decomposition | Real Partial-Systolic Q-less QR Decomposition | Real Burst QR Decomposition

Functions
`fixed.qrAB`

Topics
"Choose a Block for HDL-Optimized Fixed-Point Matrix Operations"

Real Reciprocal HDL Optimized

Compute reciprocal and generate optimized HDL code



Libraries:
Fixed-Point Designer HDL Support / Math Operations

Description

The Real Reciprocal HDL Optimized block computes 1/`u`, where `u` is a real scalar.

Limitations

Data type override is not supported for the Real Reciprocal HDL Optimized block.

Ports

Input

u — Value to take reciprocal of
real scalar

Value to take the reciprocal of, specified as a real scalar.

Slope-bias representation is not supported for fixed-point data types.

Data Types: `single` | `double` | `fixed point`

validIn — Whether input is valid
`Boolean` scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the `u` input port is valid. When this value is `1` (`true`), the block captures the value at the `u` input port. When this value is `0` (`false`), the block ignores the input samples.

Data Types: `Boolean`

Output

y — Reciprocal
real scalar

Reciprocal, returned as a real scalar with the data type specified by the **Output datatype** parameter.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether output data is valid, returned as a Boolean scalar. When the value of this control signal is `1` (`true`), the block has successfully computed the output at port `y`. When this value is `0` (`false`), the output data is not valid.

Data Types: `Boolean`

Parameters

Output datatype — Data type of output
`fixdt(1,18,10)` (default) | `single` | `fixdt(1,16,0)` | `<data type expression>`

Data type of the output `y`, specified as `fixdt(1,18,10)`, `single`, `fixdt(1,16,0)`, or as a user-specified data type expression. The type can be specified directly, or expressed as a data type object, such as `Simulink.NumericType`.

Programmatic Use
Block Parameter: `OutputType`
Type: character vector
Values: `'fixdt(1,18,10)'` | `'single'` | `'fixdt(1,16,0)'` | `'<data type expression>'`
Default: `'fixdt(1,18,10)'`

Algorithms

Division by Zero Behavior

For fixed-point input `u`, the Real Reciprocal HDL Optimized block wraps on overflow for division by zero. The behavior for fixed-point division by zero is summarized in the table below.

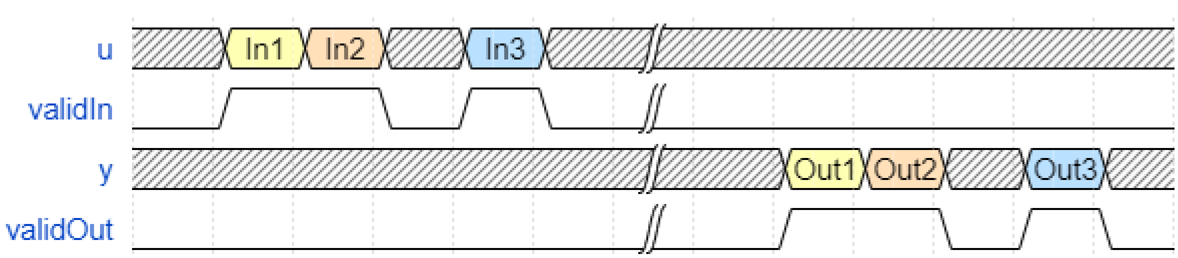
Wrap Overflow	Saturate Overflow
0/0 = 0	0/0 = 0
1/0 = 0	1/0 = upper bound
-1/0 = 0	-1/0 = lower bound

For floating-point inputs, the Real Reciprocal HDL Optimized block follows IEEE Standard 754.

How to Interface with the Real Reciprocal HDL Optimized Block

Because of its fully pipelined nature, the Real Reciprocal HDL Optimized block is able to accept input data on any cycle, including consecutive clock cycles. To send input data to the block, the `validIn` signal must be true. When the block has finished the computation and is ready to send the output, it will change `validOut` to true for one clock cycle. For inputs set of consecutive cycles, `validOut` will also be set to true on consecutive cycles.

The latency is defined from the input to the corresponding output. For example in the figure below, from `In1` to `Out1`, `In2` to `Out2`, `In3` to `Out3`, etc.



The latency depends on the input data type, as summarized in the table.

Input Type	Latency
`fi`	`nextpow2(u.WordLength) + 1 + u.WordLength + 2 - issigned(u) + 7`
Scaled double	`u.WordLength + 2 - issigned(u) + 7`
Floating point	`7`

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Slope-bias representation is not supported for fixed-point data types.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Restrictions

Supports fixed-point data types only.

See Also

Blocks

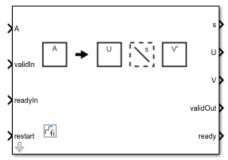
Real Divide HDL Optimized | Complex Divide HDL Optimized | Normalized Reciprocal HDL Optimized

Functions

`fixed.cordicReciprocal` | `fixed.cordicDivide`

Square Jacobi SVD HDL Optimized

Fixed-point singular value decomposition



Libraries:
Fixed-Point Designer HDL Support / Matrices and Linear Algebra / Matrix Factorizations

Description

Use the Square Jacobi SVD HDL Optimized block to perform singular value decomposition (SVD) on square matrices using the two-sided Jacobi algorithm. Given a square matrix *A*, the Square Jacobi SVD HDL Optimized block uses the two-sided Jacobi method to produce a vector *s* of nonnegative elements and unitary matrices *U* and *V* such that `A = U*diag(s)*V'`.

Ports

Input

A — Square matrix *A*
real or complex square matrix

Square n-by-n matrix *A*, specified as a real or complex square matrix.

If the matrix *A* has a fixed-point data type, the matrix must use signed binary-point scaling. Slope and bias scaling and unsigned fixed-point types are not supported.

Data Types: `single` | `double` | `fixed point`

validIn — Whether input is valid
Boolean scalar

Whether input is valid, specified as a Boolean scalar. This control signal indicates when the data from the A input port is valid. When this value is `1` (`true`) and the value at `ready` is `1` (`true`), the block captures the values at the A input port. When this value is `0` (`false`), the block ignores the input samples.

After sending a `true` `validIn` signal, a delay may occur before the `ready` signal is `false`. To ensure all data is processed, you must wait until the `ready` signal is `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

readyIn — Whether downstream block is ready
Boolean scalar

Whether downstream block is ready, specified as a `Boolean` scalar. This control signal monitors the `ready` port of the downstream block. When the `readyIn` value is `1` (`true`), and the value at `validOut` is `1` (`true`), the block outputs data to the downstream block. When the `readyIn` value is `0` (`false`), the downstream block is not ready to accept data. The Square Jacobi SVD HDL Optimized block pauses on the output stage and the `ready` signal remains `0` (`false`) until the `readyIn` signal is high.

Data Types: `Boolean`

restart — Whether to clear internal states
Boolean scalar

Whether to clear internal states, specified as a Boolean scalar. When this value is 1 (`true`), the block stops the current calculation and clears all internal states. When this value is 0 (`false`) and the `validIn` value is 1 (`true`), the block begins a new subframe.

Data Types: `Boolean`

Output

s — Singular values
column vector

Singular values, returned as a column vector of length `n`. The "Singular Values" are nonnegative and returned in descending order such that `s(1) >= s(2) >= ... >= s(n)`. Singular values are returned with the same data type as the input matrix `A`.

Data Types: `single` | `double` | `fixed point`

U — Left singular vectors
unitary `n`-by-`n` matrix

Left singular vectors, returned as a unitary `n`-by-`n` matrix.

For fixed-point and scaled-double inputs, `U` is returned as a signed fixed-point or scaled-double `fi` with the same word length as `A` and fraction length equal to two less than the word length. One of these integer bits is used for the sign. The other integer bit allows `+1` to be represented exactly.

For floating-point input, `U` has the same data type as `A`.

Data Types: `single` | `double` | `fixed point`

V — Right singular vectors
unitary `n`-by-`n` matrix

Right singular vectors, returned as a unitary `n`-by-`n` matrix.

For fixed-point and scaled-double inputs, `V` is returned as a signed fixed-point or scaled-double `fi` with the same word length as `A` and fraction length equal to two less than the word length. One of these integer bits is used for the sign. The other integer bit allows `+1` to be represented exactly.

For floating-point input, `V` has the same data type as `A`.

Data Types: `single` | `double` | `fixed point`

validOut — Whether output data is valid
`Boolean` scalar

Whether the output data is valid, returned as a Boolean scalar. This control signal indicates when the data at the output ports `U`, `S`, and `V` are valid. When this value is `1` (`true`), the output data is valid.

When this value is `0` (`false`), the output data is not valid. Transfer of data to the downstream block occurs only when both the `validOut` and `readyIn` signals are high.

Data Types: `Boolean`

ready — Whether block is ready
`Boolean` scalar

Whether the block is ready, returned as a Boolean scalar. This control signal indicates when the block is ready for new input data. When this value is `1` (`true`) and the `validIn` value is `1` (`true`), the block accepts input data in the next time step. When this value is `0` (`false`), the block ignores input data in the next time step.

After sending a `true` `validIn` signal, there may be some delay before the `ready` signal is `false`. To ensure all data is processed, you must wait until the `ready` signal is `false` before sending another `true` `validIn` signal.

Data Types: `Boolean`

Parameters

Dimension of matrix A — Dimension of square matrix *A*
`n` (default) | positive integer-valued scalar

Dimension of square matrix *A*, specified as a positive integer-valued scalar.

Programmatic Use
Block Parameter: `n`
Type: `string`
Values: positive integer-valued scalar
Default: `n`

Number of Jacobi iterations — Number of iterations of Jacobi algorithm
`10` (default) | positive integer

Number of iterations of the Jacobi algorithm, specified as a positive integer. Most sources indicate that 10 iterations is sufficient for the Jacobi algorithm to converge [7][8][9][10].

Programmatic Use
Block Parameter: `nIterations`
Type: `string`
Values: positive integer-valued scalar
Default: `10`

Select outputs — Block outputs
`UsV` (default) | `Us` | `sV` | `s`

Block outputs, specified as `UsV`, `Us`, `sV`, or `s`.

Programmatic Use
Block Parameter: `outputOption`
Type: `string`
Values: `UsV` | `Us` | `sV` | `s`
Default: `UsV`

Signal type — Complexity of input matrix *A*
`real` (default) | `complex`

Complexity of the input matrix *A*, specified as `real`, or `complex`.

Programmatic Use
Block Parameter: `complexity`
Type: `string`
Values: `real` | `complex`
Default: `real`

Tips

- The Square Jacobi SVD HDL Optimized block computes the singular value decomposition in place. Set the fixed-point data type of the input `n`-by-`n` matrix `A` with enough precision and enough headroom to avoid overflow.

  First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Then define the integer length based on the value of the upper bound, with one additional bit for the sign, another additional bit for intermediate CORDIC growth, and one more bit for intermediate growth to compute the Jacobi rotations. Compute the fraction length based on the integer length and the desired word length.

  ```
  svdUpperBound = fixed.singularValueUpperBound(n,n,max(abs(A(:))))
  additionalBitGrowth = 3;
  integerLength = ceil(log2(svdUpperBound)) + additionalBitGrowth
  wordLength = 16
  fractionLength = wordLength - integerLength
  ```
- The behavior of the Square Jacobi SVD HDL Optimized block is equivalent to `[U,s,V] = fixed.jacobiSVD(A)`. The `fixed.JacobiSVD` function uses the same algorithm as the Square Jacobi SVD HDL Optimized, with the same output data types. However, there may be small numerical differences in the least-significant bit between the function and the block.
- The behavior of the Square Jacobi SVD HDL Optimized block is equivalent to `[U,s,V] = fixed.svd(A,'econ','vector')` when `A` is a square matrix.

Algorithms

Jacobi Singular Value Decomposition

The Square Jacobi SVD HDL Optimized block uses a two-sided Jacobi algorithm for singular value decomposition (SVD) [2][3][4]. Compared to the sequential Golub-Kahan-Reinsch algorithm for SVD [5], the Jacobi algorithm has inherent parallelism and performs better for FPGA and ASIC applications [6]. The Jacobi method is an iterative algorithm. Use the **Number of Jacobi iterations** parameter to set the number of iterations necessary for convergence. Most sources indicate that 10 iterations is sufficient for the Jacobi algorithm to converge [7][8][9][10].

AMBA AXI Handshake Process

This block uses the AMBA AXI handshake protocol on both the input and the output side [1]. The `valid/ready` handshake process is used to transfer data and control information. This two-way control mechanism allows both the manager and subordinate to control the rate at which information moves between manager and subordinate.

On the input side, the validIn signal indicates when the upstream data is available. The ready signal indicates that the block can accept the data. Transfer of data from the upstream block occurs only when both the validIn and ready signals are high.

On the output side, the validOut signal indicates the solution data is available. The readyIn signal indicates when the downstream block can accept data. Transfer of data to the downstream block occurs only when both the validOut and readyIn signals are high.

When the readyIn signal is low, the block pauses on the output stage and the ready signal remains low. This configuration allows a stall from the downstream block to back-propagate to the upstream block.

To use the Square Jacobi SVD HDL Optimized block in feed-forward fashion without back pressure from the downstream block, feed a constant Boolean 'true' to the readyIn port.

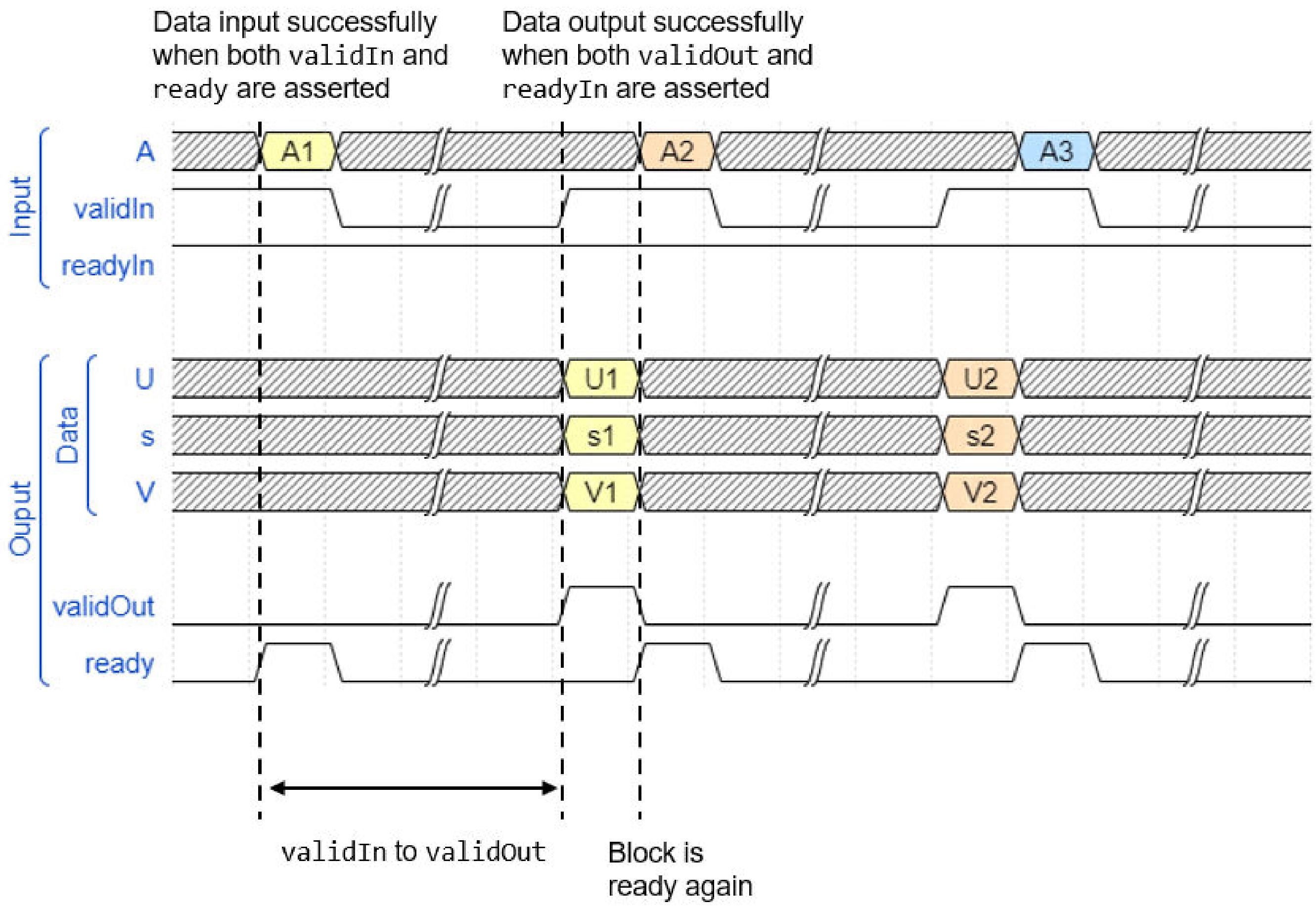
Block Timing

The latency of the Square Jacobi SVD HDL Optimized block depends on the size (n), complexity, and word length (wl) of the input matrix A, and the number of iterations (nIterations) of the two-sided Jacobi algorithm, as summarized in the table.

- If the data type of *A* is fixed point, then wl is the word length.
- If the data type of *A* is double precision, then wl is 53.
- If the data type of *A* is single precision, then wl is 24.

Signal Complexity	validIn to validOut
Real	(wl*2+31)*(n-1+rem(n,2))*nIterations+2+nextpow2(n)*(n
Complex	(wl*6+48)*(n-1+rem(n,2))*nIterations+2+nextpow2(n)*(n

For example, assume that validIn asserts before ready, meaning that the upstream data source is faster than the Square Jacobi SVD HDL Optimized block. Additionally, assume that readyIn is always asserted, meaning that the downstream consumer of the data is faster than the Square Jacobi SVD HDL Optimized.



In the figure:

- `A1` is the first *A* matrix, `U1` is the first *U* matrix, `s1` is the first *s* vector, `V1` is the first *V* matrix, and so on.
- `validIn` to `validOut` — Goes from a successful matrix input to the block starting to output the solution.
- After a successful solution output, the block is ready again at the next clock cycle.

Hardware Resource Utilization

This block supports HDL code generation using the Simulink HDL Workflow Advisor. For an example, see "HDL Code Generation and FPGA Synthesis from Simulink Model" (HDL Coder) and "Implement Digital Downconverter for FPGA" (DSP HDL Toolbox).

This example data was generated by synthesizing the block on a Xilinx Zynq-7000 xc7z100 SoC. The synthesis tool was Vivado v2022.1 (win64).

Signal Type: Real

The following parameters were used for synthesis.

- Block parameters:

  - `n = 8`
  - `nIterations = 10`
- Input data type: `sfix32_En25`
- Target frequency: 90 MHz

These tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization
Slice LUTs	71541	277400	25.79%
Slice Registers	35781	554800	6.45%
DSPs	256	2020	12.67%
Block RAM Tile	0	755	0%

	Value
Requirement	11.111 ns (90 MHz)
Data Path Delay	11.007 ns
Slack	0.085 ns
Clock Frequency	90.69 MHz

Signal Type: Complex

The following parameters were used for synthesis.

- Block parameters:
  - `n = 8`
  - `nIterations = 10`
- Input data type: `sfix32_En25`
- Target frequency: 60 MHz

The following tables show the post place-and-route resource utilization results and timing summary, respectively.

Resource	Usage	Available	Utilization
Slice LUTs	196506	277400	70.84%
Slice Registers	85150	554800	15.35%
DSPs	1344	2020	66.53%
Block RAM Tile	0	755	0%

	Value
Requirement	16.667 ns (60 MHz)
Data Path Delay	16.572 ns
Slack	0.076 ns

	Value
Clock Frequency	60.27 MHz

Version History

Introduced in R2023a

References

[1] Arm Developer. "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process.

[2] Jacobi, Carl G. J., "Über ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen." *Journal fur die reine und angewandte Mathematik* 30 (1846): 51–94.

[3] Forsythe, George E. and Peter Henrici. "The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix." *Transactions of the American Mathematical Society* 94, no. 1 (January 1960): 1-23.

[4] Shiri, Aidin and Ghader Khosroshahi. "An FPGA Implementation of Singular Value Decomposition", *ICEE 2019: 27th Iranian Conference on Electrical Engineering, Yazd, Iran, April 30–May 2, 2019*, 416-22, IEEE.

[5] Golub, Gene H. and Charles F. Van Loan. *Matrix Computations*, 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013.

[6] Athi, Mrudula V., Seyed R. Zekavat, and Alan A. Struthers. "Real-Time Signal Processing of Massive Sensor Arrays via a Parallel Fast Converging SVD Algorithm: Latency, Throughput, and Resource Analysis." *IEEE Sensors Journal* 16, no. 18 (January 2016): 2519-26. https://doi.org/10.1109/JSEN.2016.2517040.

[7] Brent, Richard P., Franklin T. Luk, and Charles Van Loan. "Computation of the Singular Value Decomposition Using Mesh-Connected Processors." *Journal of VLSI and Computer Systems* 1, 3 (1985): 242–70.

[8] Hemkumar, Nariankadu D. *A Systolic VLSI Architecture for Complex SVD*. Master’s thesis, Rice University, 1991.

[9] Duryea, R. A. *Finite Precision Arithmetic in Singular Value Decomposition Architectures*. Ph.D. thesis, Cornell University, 1987.

[10] Cavallaro, Joseph R. and Franklin T. Luk. 1987. "CORDIC Arithmetic for an SVD Processor." *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH), Como, Italy, May 18-21, 1987*, 113-20. IEEE. https://doi.org/10.1109/ARITH.1987.6158686.

Extended Capabilities

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Note When generating code for large matrices set the configuration parameter **HDL Code Generation > Global Settings > Ports** tab **Check for DUT pin count exceeding I/O Threshold** to `Warning` or `None`. To learn more about this setting, see “DUT Port Configuration Parameters” (HDL Coder).

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has one default HDL architecture.

HDL Block Properties

General	
ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

Supports fixed-point data types only. Fixed-point data types must use signed binary-point scaling. Slope and bias scaling and unsigned fixed-point types are not supported.

See Also

“Singular Values” | `fixed.svd`

Properties

fi Object Data Properties

Data properties of the `fi` object

Description

The `fi` object has three types of properties: data properties, `fimath` properties, and `numerictype` properties. Use data properties to access data in a `fi` object using dot notation. The data properties of a `fi` object are always writable.

Properties

Data Properties

`bin` — Stored integer value of `fi` object
binary

Stored integer value of `fi` object in binary.

`data` — Numerical real-world value of `fi` object
double

Numerical real-world value of `fi` object.

`dec` — Stored integer value of `fi` object
decimal

Stored integer value of `fi` object in decimal.

`double` — Real-world value of `fi` object
double

Real-world value of `fi` object stored as a MATLAB double.

`hex` — Stored integer value of `fi` object
hexadecimal

Stored integer value of `fi` object in hexadecimal.

`int` — Stored integer value of `fi` object
integer

Stored integer value of `fi` object stored in a built-in MATLAB integer data type.

`oct` — Stored integer value of `fi` object
octal

Stored integer value of `fi` object in octal.

`Value` — Full-precision real-world value of `fi` object
character vector

Full-precision real-world value of `fi` object, stored as a character vector.

Version History

Introduced before R2006a

See Also

`fi`

fimath Object Properties

Properties of the `fimath` object

Description

`fimath` properties determine the rules for performing fixed-point arithmetic operations on `fi` objects. `fimath` properties are, by transitivity, also properties of the `fi` object. You can set these properties for individual `fi` objects. `fimath` object properties are always writable.

Properties

Sum Data Type Properties

`CastBeforeSum` — Whether both operands are cast to the sum data type before addition
`0` (`false`) (default) | `1` (`true`)

Whether both operands are cast to the sum data type before addition, specified as one of these values:

- `0` (`false`) — Do not cast before sum
- `1` (`true`) — Cast before sum

Note This property is hidden when the `SumMode` is set to `FullPrecision`.

`MaxSumWordLenth` — Maximum allowable word length for the sum data type
`65535` (default) | positive integer

Maximum allowable word length for the sum data type, specified as a positive integer.

`SumBias` — Bias of the sum data type
`0` (default) | floating-point number

Bias of the sum data type, specified as a floating-point number.

`SumFixedExponent` — Fixed exponent of the sum data type
`-30` (default) | positive or negative integer

Fixed exponent of the sum data type, specified as a positive or negative integer.

Note The `SumFractionLength` is the negative of the `SumFixedExponent`. Changing one property changes the other.

`SumFractionLength` — Fraction length of the sum data type in bits
`30` (default) | positive or negative integer

Fraction length of the sum data type in bits, specified as a positive or negative integer.

Note The `SumFractionLength` is the negative of the `SumFixedExponent`. Changing one property changes the other.

`SumMode` — How the sum data type is determined

`FullPrecision` (default) | `KeepLSB` | `KeepMSB` | `SpecifyPrecision`

How the sum data type is determined, specified as one of the following:

- `FullPrecision` — Keep the full precision of the result.
- `KeepLSB` — Keep least significant bits. Specify the sum data type word length, while the fraction length is set to maintain the least significant bits of the sum.
- `KeepMSB` — Keep most significant bits. Specify the sum data type word length, while the fraction length is set to maintain the most significant bits of the sum and no more fractional bits than necessary.
- `SpecifyPrecision` — Specify the word and fraction lengths or the slope and bias of the sum data type.

`SumSlope` — Slope of the sum data type

`9.3132e-010` (default) | floating-point number

Slope of the sum data type, specified as a floating-point number.

Note

$$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$$

Changing one of these properties affects the others.

`SumSlopeAdjustmentFactor` — Slope adjustment factor of the sum data type

`1` (default) | floating-point number greater than or equal to 1 and less than 2

Slope adjustment factor of the sum data type, specified as a floating-point number greater than or equal to 1 and less than 2.

Note

$$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$$

Changing one of these properties affects the others.

`SumWordLength` — Word length of the sum data type in bits

`32` (default) | positive integer

Word length of the sum data type in bits, specified as a positive integer.

Product Data Type Properties

`MaxProductWordLenth` — Maximum allowable word length for the product data type

`65535` (default) | positive integer

Maximum allowable word length for the product data type, specified as a positive integer.

ProductBias — Bias of the product data type
0 (default) | floating-point number

Bias of the product data type, specified as a floating-point number.

ProductFixedExponent — Fixed exponent of the product data type
-30 (default) | positive or negative integer

Fixed exponent of the product data type, specified as a positive or negative integer.

Note The ProductFractionLength is the negative of the ProductFixedExponent. Change one property changes the other.

ProductFractionLength — Fraction length of the product data type in bits
30 (default) | positive or negative integer

Fraction length of the product data type in bits, specified as a positive or negative integer.

Note The ProductFractionLength is the negative of the ProductFixedExponent. Changing one property changes the other.

ProductMode — How the product data type is determined
FullPrecision (default) | KeepLSB | KeepMSB | SpecifyPrecision

How the product data type is determined, specified as one of these values:

- FullPrecision — Keep the full precision of the result.
- KeepLSB — Keep least significant bits. Specify the product word length, while the fraction length is set to maintain the least significant bits of the product.
- KeepMSB — Keep most significant bits. Specify the product word length, while the fraction length is set to maintain the most significant bits of the product.
- SpecifyPrecision — Specify the word and fraction lengths or slope and bias of the product.

ProductSlope — Slope of the product data type
9.3132e-010 (default) | floating-point number

Slope of the product data type, specified as a floating-point number.

Note

$$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$$

Changing one of these properties affects the others.

ProductSlopeAdjustmentFactor — Slope adjustment factor of the product data type
1 (default) | floating-point number greater than or equal to 1 and less than 2

Slope adjustment factor of the product data type, specified as a floating-point number greater than or equal to 1 and less than 2.

Note

$$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$$

Changing one of these properties affects the others.

`ProductWordLength` — Word length of the product data type in bits
32 (default) | positive integer

Word length of the product data type in bits, specified as a positive integer.

Rounding and Overflow Properties

`OverflowAction` — Action to take on overflow
`Saturate` (default) | `Wrap`

Action to take on overflow, specified as one of these values:

- `Saturate` — Saturate to maximum or minimum value of the fixed-point range on overflow.
- `Wrap` — Wrap on overflow. This mode is also known as two's complement overflow.

`RoundingMethod` — Rounding method to use
`Nearest` (default) | `Ceiling` | `Convergent` | `Zero` | `Floor` | `Round`

Rounding method to use, specified as one of the following:

- `Nearest` — Round toward nearest. Ties round toward positive infinity.
- `Ceiling` — Round toward positive infinity.
- `Convergent` — Round toward nearest. Ties round to the nearest even stored integer (least biased).
- `Zero` — Round toward zero.
- `Floor` — Round toward negative infinity.
- `Round` — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

Version History
Introduced before R2006a

numerictype Object Properties

Properties of the `numerictype` object

Description

`numerictype` object properties define the data type and scaling attributes of a fixed-point object. All properties of a `numerictype` object are writable. However, the `numerictype` properties of a `fi` object become read only after the `fi` object has been created. Any `numerictype` properties of a `fi` object that are unspecified at the time of `fi` object creation are automatically set to their default values.

Properties

numerictype Object Properties

`Bias` — Bias associated with the object
`0` (default) | floating-point number

Bias associated with the object, specified as a floating-point number. Along with the slope, the bias forms the scaling of a fixed-point number.

`DataType` — Data type category
`Fixed` (default) | `double` | `single` | `boolean` | `ScaledDouble`

Data type category, specified as one of the following:

- `Fixed` — Fixed-point or integer data type
- `double` — Built-in MATLAB `double` data type
- `single` — Built-in MATLAB `single` data type
- `boolean` — Built-in MATLAB `boolean` data type
- `ScaledDouble` — Scaled double data type

`DataTypeMode` — Data type and scaling associated with the object
`Fixed-point: binary point scaling` (default) | `Fixed-point: slope and bias scalingFixed-point: unspecified scaling` | `Scaled double: binary point scaling` | `Scaled double: slope and bias scaling` | `Scaled double: unspecified scaling` | `Double` | `Single` | `Boolean`

Data type and scaling associated with the object, specified as one of the following:

- `Fixed-point: binary point scaling` — Fixed-point data type and scaling defined by the word length and fraction length
- `Fixed-point: slope and bias scaling` — Fixed-point data type and scaling defined by the slope and bias
- `Fixed-point: unspecified scaling` — Fixed-point data type with unspecified scaling
- `Scaled double: binary point scaling` — Double data type with fixed-point word length and fraction length information retained

- `Scaled double: slope and bias scaling` — Double data type with fixed-point slope and bias information retained
- `Scaled double: unspecified scaling` — Double data type with unspecified fixed-point scaling
- `Double` — Built-in `double` data type
- `Single` — Built-in `single` data type
- `Boolean` — Built-in `boolean` data type

For more details on these data types, see "Valid Values for numerictype Object Properties".

`FixedExponent` — Fixed-point exponent associated with the object
`-15` (default) | integer

Fixed-point exponent associated with the object, specified as an integer.

Note The `FixedExponent` property is the negative of the `FractionLength`. Changing one property changes the other.

`FractionLength` — Fraction length of the stored integer value in bits
best precision fraction length (default) | integer

Fraction length of the stored integer value in bits, specified as an integer. The default is the best precision fraction length based on the value of the object and the word length.

Note The `FractionLength` property is the negative of the `FixedExponent`. Changing one property changes the other.

`Scaling` — Scaling mode of the object
`BinaryPoint` (default) | `SlopeBias` | `Unspecified`

Scaling mode of the object, specified as one of the following:

- `BinaryPoint` (default) — Scaling for the `fi` object is defined by the fraction length.
- `SlopeBias` — Scaling for the `fi` object is defined by the slope and bias.
- `Unspecified` — Unspecified scaling. This is a temporary setting that is only allowed at `fi` object creation to allow for the automatic assignment of a binary point best-precision scaling.

`Signed` — Whether the object is signed (not recommended)
`1` (`true`) (default) | `0` (`false`) | `[]`

Whether the object is signed, specified as one of the following:

- `1` (`true`) — Signed
- `0` (`false`) — Unsigned
- `[]` — Auto

Note The `Signed` property is not recommended. Use `Signedness` instead. There are no plans to remove the `Signed` property.

Although the Signed property is still supported, the Signedness property always appears in the numerictype object display. If you choose to change or set the signedness of your numerictype objects using the Signed property, MATLAB updates the corresponding value of the Signedness property.

Signedness — Whether the object is signed

Signed (default) | Unsigned | Auto

Whether the object is signed, specified as one of the following:

- Signed — Signed
- Unsigned — Unsigned
- Auto — Unspecified sign

Note numerictype objects can have a Signedness of Auto, but all fi objects must be Signed or Unsigned. If a fi object with Auto Signedness is used to create a fi object, the Signedness property of the fi object automatically defaults to Signed.

Slope — Slope associated with the object

finite floating-point number greater than zero

Slope associated with the object, specified as a finite floating-point number greater than zero. Along with the bias, the slope forms the scaling of a fixed-point number.

Note $Slope = SlopeAdjustmentFactor \text{ X } 2^{FixedExponent}$Changing one of these properties changes the other.

SlopeAdjustmentFactor — Slope adjustment associated with the object

number greater than or equal to 1 and less than 2

Slope adjustment associated with the object, specified as a number greater than or equal to 1 and less than 2.

Note $Slope = SlopeAdjustmentFactor \text{ X } 2^{FixedExponent}$Changing one of these properties changes the other.

WordLength — Word length of the stored integer value in bits

16 (default) | positive integer if Signedness is Unsigned or unspecified | integer greater than one if Signedness is set to Signed

Word length of the stored integer value in bits, specified as:

- A positive integer if Signedness is Unsigned or unspecified
- An integer greater than one if Signedness is set to Signed

Version History

Introduced before R2006a

4

Functions

abs

Absolute value of `fi` object

Syntax

```
y = abs(a)
y = abs(a,T)
y = abs(a,F)
y = abs(a,T,F)
```

Description

`y = abs(a)` returns the absolute value of `fi` object `a` with the same `numerictype` object as `a`. Intermediate quantities are calculated using the `fimath` associated with `a`. The output `fi` object, `y`, has the same local `fimath` as `a`.

`y = abs(a,T)` returns a `fi` object with a value equal to the absolute value of `a` and `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` associated with `a` and the output `fi` object `y` has the same local `fimath` as `a`. See “Data Type Propagation Rules” on page 4-8.

`y = abs(a,F)` returns a `fi` object with a value equal to the absolute value of `a` and the same `numerictype` object as `a`. Intermediate quantities are calculated using the `fimath` object `F`. The output `fi` object, `y`, has no local `fimath`.

`y = abs(a,T,F)` returns a `fi` object with a value equal to the absolute value of `a` and the `numerictype` object `T`. Intermediate quantities are calculated using the `fimath` object `F`. The output `fi` object, `y`, has no local `fimath`. See “Data Type Propagation Rules” on page 4-8.

Examples

Absolute Value of Most Negative Representable Value

This example shows the difference between the absolute value results for the most negative value representable by a signed data type when the `'OverflowAction'` property is set to `'Saturate'` or `'Wrap'`.

Calculate the absolute value when the `'OverflowAction'` is set to the default value `'Saturate'`.

```
P = fipref('NumericTypeDisplay','full',...
           'FimathDisplay','full');
a = fi(-128)
y = abs(a)
```

```
a = 

  -128

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
```

```
        FractionLength: 8

y = 

  127.9961

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8
```

abs returns 127.9961, which is a result of saturation to the maximum positive value.

Calculate the absolute value when the 'OverflowAction' is set to 'Wrap'.

```
a.OverflowAction = 'Wrap'
y = abs(a)
```

```
a = 

  -128

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision

y = 

  -128

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

abs returns 128, which is a result of wrapping back to the most negative value.

Difference Between Absolute Values for Real and Complex fi Inputs

This example shows the difference between the absolute value results for complex and real fi inputs that have the most negative value representable by a signed data type when the 'OverflowAction' property is set to 'Wrap'.

Define a complex fi object.

```
re = fi(-1,1,16,15);
im = fi(0,1,16,15);
a = complex(re,im)
```

```
a =

  -1.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

`a` is complex, but numerically equal to the real part, `re`.

Calculate the absolute value of the complex `fi` object.

```
y = abs(a,re.numerictype,fimath('OverflowAction','Wrap'))
```

```
y =

    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Calculate the absolute value of the real `fi` object.

```
y = abs(re,re.numerictype,fimath('OverflowAction','Wrap'))
```

```
y =

    -1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Specify `numerictype` and `fimath` Inputs to Control the Result of `abs` for Real Inputs

This example shows how to specify `numerictype` and `fimath` objects as optional arguments to control the result of the `abs` function for real inputs. When you specify a `fimath` object as an argument, that `fimath` object is used to compute intermediate quantities, and the resulting `fi` object has no local `fimath`.

```
a = fi(-1,1,6,5,'OverflowAction','Wrap');
y = abs(a)
```

```
y =

    -1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
        WordLength: 6
    FractionLength: 5

    RoundingMethod: Nearest
    OverflowAction: Wrap
       ProductMode: FullPrecision
           SumMode: FullPrecision
```

The returned output is identical to the input. This may be undesirable because the absolute value is expected to be positive.

```
F = fimath('OverflowAction','Saturate');
y = abs(a,F)
```

```
y =

    0.9688

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 6
    FractionLength: 5
```

The returned `fi` object is saturated to a value of `0.9688` and has the same `numerictype` object as the input.

Because the output of `abs` is always expected to be positive, an unsigned `numerictype` may be specified for the output.

```
T = numerictype(a.numerictype, 'Signed', false);
y = abs(a,T,F)
```

```
y =

     1

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Unsigned
        WordLength: 6
    FractionLength: 5
```

Specifying an unsigned `numerictype` enables better precision.

Specify numerictype and fimath Inputs to Control the Result of abs for Complex Inputs

This example shows how to specify `numerictype` and `fimath` objects as optional arguments to control the result of the `abs` function for complex inputs.

Specify a `numerictype` input and calculate the absolute value of `a`.

```
a = fi(-1-i,1,16,15,'OverflowAction','Wrap');
T = numerictype(a.numerictype,'Signed',false);
y = abs(a,T)
```

```
y =
```

```
    1.4142

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 15

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

A `fi` object is returned with a value of `1.4142` and the specified unsigned `numerictype`. The `fimath` used for intermediate calculation and the `fimath` of the output are the same as that of the input.

Now specify a `fimath` object different from that of `a`.

```
F = fimath('OverflowAction','Saturate','SumMode',...
        'KeepLSB','SumWordLength',a.WordLength,...
        'ProductMode','specifyprecision',...
        'ProductWordLength',a.WordLength,...
        'ProductFractionLength',a.FractionLength);
y = abs(a,T,F)

y =

    1.4142

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 15
```

The specified `fimath` object is used for intermediate calculation. The `fimath` associated with the output is the default `fimath`.

Input Arguments

a — Input fi array

scalar | vector | matrix | multidimensional array

Input `fi` array, specified as a scalar, vector, matrix, or multidimensional array.

`abs` only supports `fi` objects with trivial [Slope Bias] scaling, that is, when the bias is `0` and the fractional slope is `1`.

`abs` uses a different algorithm for real and complex inputs. For more information, see “Absolute Value” on page 4-7.

Data Types: `fi`
Complex Number Support: Yes

T — numerictype of the output

`numerictype` object

`numerictype` of the output `fi` object `y`, specified as a `numerictype` object. For more information, see "Data Type Propagation Rules" on page 4-8.

Example: `T = numerictype(0,24,12,'DataType','Fixed')`

F — Fixed-point math settings to use
`fimath` object

Fixed-point math settings to use for the calculation of absolute value, specified as a `fimath` object.

Example: `F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent')`

Algorithms

Absolute Value

The absolute value of a real number is the corresponding nonnegative value that disregards the sign.

For a real input, `a`, the absolute value, `y`, is:

y = a if a >= 0 (4-1)

y = -a if a < 0 (4-2)

`abs(-0)` returns `0`.

Note When the `fi` object `a` is real and has a signed data type, the absolute value of the most negative value is problematic since it is not representable. In this case, the absolute value saturates to the most positive value representable by the data type if the `'OverflowAction'` property is set to `'Saturate'`. If `'OverflowAction'` is `'Wrap'`, the absolute value of the most negative value has no effect.

For a complex input, `a`, the absolute value, `y`, is related to its real and imaginary parts as follows:

`y = sqrt(real(a)*real(a) + imag(a)*imag(a))` (4-3)

The `abs` function computes the absolute value of a complex input, `a`, as follows:

1. Calculate the real and imaginary parts of `a`.

   `re = real(a)` (4-4)

   `im = imag(a)` (4-5)
2. Compute the squares of `re` and `im` using one of the following objects:
   - The `fimath` object `F` if `F` is specified as an argument.
   - The `fimath` associated with `a` if `F` is not specified as an argument.
3. If the input is signed, cast the squares of `re` and `im` to unsigned types.
4. Add the squares of `re` and `im` using one of the following objects:
   - The `fimath` object `F` if `F` is specified as an argument.
   - The `fimath` object associated with `a` if `F` is not specified as an argument.

5 Compute the square root of the sum computed in Step 4 using the `sqrt` function with the following additional arguments:

- The `numerictype` object `T` if `T` is specified, or the `numerictype` object of `a` otherwise.
- The `fimath` object `F` if `F` is specified, or the `fimath` object associated with `a` otherwise.

Note Step 3 prevents the sum of the squares of the real and imaginary components from being negative. This is important because if either `re` or `im` has the maximum negative value and the `'OverflowAction'` property is set to `'Wrap'` then an error will occur when taking the square root in Step 5.

Data Type Propagation Rules

For syntaxes for which you specify a `numerictype` object `T`, the `abs` function follows the data type propagation rules listed in the following table. In general, these rules can be summarized as “floating-point data types are propagated.” This allows you to write code that can be used with both fixed-point and floating-point inputs.

Data Type of Input `fi` Object `a`	Data Type of `numerictype` object `T`	Data Type of Output `y`
`fi` `Fixed`	`fi` `Fixed`	Data type of `numerictype` object `T`
`fi` `ScaledDouble`	`fi` `Fixed`	`ScaledDouble` with properties of `numerictype` object `T`
`fi` `double`	`fi` `Fixed`	`fi` `double`
`fi` `single`	`fi` `Fixed`	`fi` `single`
Any `fi` data type	`fi` `double`	`fi` `double`
Any `fi` data type	`fi` `single`	`fi` `single`

Note When the `Signedness` of the input `numerictype` object `T` is `Auto`, the `abs` function always returns an `Unsigned` `fi` object.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Double and complex data types are not supported.

See Also

fi | fimath | numerictype

accumneg

Subtract two `fi` objects or values

Syntax

```
c = accumneg(a,b)
c = accumneg(a,b,RoundingMethod)
c = accumneg(a,b,RoundingMethod,OverflowAction)
```

Description

`c = accumneg(a,b)` subtracts `b` from `a` using the data type of `a`. `b` is cast into the data type of `a`. If `a` is a fi object, the default `'Floor'` rounding method and default `'Wrap'` overflow action are used. The `fimath` properties of `a` and `b` are ignored.

`c = accumneg(a,b,RoundingMethod)` subtracts `b` from `a` using the rounding method specified by `RoundingMethod` if `a` is a `fi` object.

`c = accumneg(a,b,RoundingMethod,OverflowAction)` subtracts `b` from `a` using the rounding method specified by `RoundingMethod` and the overflow action specified by `OverflowAction` if `a` is a `fi` object.

Examples

Subtract Two `fi` Objects or Values

This example shows how to subtract two `fi` numbers using `accumneg`.

Subtract two `fi` numbers

Subtract `b` from `a`, where `a` and `b` are both `fi` numbers, using the default rounding method of `'Floor'` and overflow action of `'Wrap'`.

```
a = fi(pi,1,16,13);
b = fi(1.5,1,16,14);
subtr_default = accumneg(a,b)
```

```
subtr_default =
    1.6416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Subtract two `fi` numbers using specified rounding and overflow action

Subtract `b` from `a`, where `a` and `b` are both `fi` numbers, using specified rounding method of `'Nearest'` and overflow action of `'Saturate'`.

```
a = fi(pi,1,16,13);
b = fi(1.5,1,16,14);
subtr_custom = accumneg(a,b,'Nearest','Saturate')
```

```
subtr_custom =
    1.6416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

a — Number to subtract from

`fi` object (default) | `double` | `single` | `logical` | `integer`

Number from which to subtract. The data type of `a` is used to compute the output data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

b — Number to subtract

`fi` object (default) | `double` | `single` | `logical` | `integer`

Number to subtract.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`RoundingMethod` — Rounding method to use

`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method to use if `a` is a `fi` object.

Example: `c = accumneg(a,b,'Ceiling')`

Data Types: `string`

`OverflowAction` — Overflow action to take

`'Wrap'` (default) | `'Saturate'`

Overflow action to take if `a` is a `fi` object.

Example: `c = accumneg(a,b,'Ceiling','Saturate')`

Data Types: `string`

Output Arguments

c — Difference of inputs

`fi` object | `double` | `single` | logical | integer

Result of subtracting input `b` from input `a`.

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
`accumpos`

Topics
"Avoid Multiword Operations in Generated Code"

accumpos

Add two fi objects or values

Syntax

```
c = accumpos(a,b)
c = accumpos(a,b,RoundingMethod)
c = accumpos(a,b,RoundingMethod,OverflowAction)
```

Description

c = accumpos(a,b) adds a and b using the data type of a. b is cast into the data type of a. If a is a fi object, the default 'Floor' rounding method and default 'Wrap' overflow action are used. The fimath properties of a and b are ignored.

c = accumpos(a,b,RoundingMethod) adds a and b using the rounding method specified by RoundingMethod.

c = accumpos(a,b,RoundingMethod,OverflowAction) adds a and b using the rounding method specified by RoundingMethod and the overflow action specified by OverflowAction.

Examples

Add Two fi Objects or Values

This example shows how to add two fi numbers using accumpos.

Add two fi numbers

Add a and b, where a and b are both fi numbers, using the default rounding method of 'Floor' and overflow action of 'Wrap'.

```
a = fi(pi,1,16,13);
b = fi(1.5,1,16,14);
add_default = accumpos(a,b)
```

```
add_default =
   -3.3584

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Add two fi numbers using specified rounding and overflow action

Add a and b, where a and b are both fi numbers, using specified rounding method of 'Nearest' and overflow action of 'Saturate'.

```
a = fi(pi,1,16,13);
b = fi(1.5,1,16,14);
add_custom = accumpos(a,b,'Nearest','Saturate')

add_custom =
    3.9999

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

a — Number to add
fi object (default) | double | single | logical | integer

Number to add. The data type of a is used to compute the output data type.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

b — Number to add
fi object (default) | double | single | logical | integer

Number to add.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

RoundingMethod — Rounding method to use
'Floor' (default) | 'Ceiling' | 'Convergent' | 'Nearest' | 'Round' | 'Zero'

Rounding method to use if a is a fi object.

Example: c = accumpos(a,b,'Ceiling')

Data Types: string

OverflowAction — Overflow action to take
'Wrap' (default) | 'Saturate'

Overflow action to take if a is a fi object.

Example: c = accumpos(a,b,'Ceiling','Saturate')

Data Types: string

Output Arguments

c — Sum of inputs
fi object | double | single | logical | integer

Result of adding input a and input b.

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`accumneg`

Topics

"Avoid Multiword Operations in Generated Code"

add

Add two `fi` objects using `fimath` object

Syntax

`c = add(F,a,b)`

Description

`c = add(F,a,b)` adds `fi` objects `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output of `fi` object `c` has no local `fimath`.

Examples

Add Two Fixed-Point Numbers

In this example, `c` is the 32-bit sum of `a` and `b` with a fraction length of 16.

```
a = fi(pi);
b = fi(exp(1));
F = fimath('SumMode','SpecifyPrecision',...
  'SumWordLength',32,'SumFractionLength',16);
c = add(F,a,b)
```

```
c =

    5.8599

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16
```

Input Arguments

F — `fimath`
`fimath` object

`fimath` object to use for addition.

`a,b` — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays.

`a` and `b` must both be `fi` objects and must have the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object.

Data Types: `fi`

Complex Number Support: Yes

Algorithms

```
c = add(F,a,b)
```

is similar to

```
a.fimath = F;
b.fimath = F;
c = a + b
```

but not identical. When you use `add`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object, `c`, has no local `fimath`. When you use the syntax `c = a + b`, where `a` and `b` have their own `fimath` objects, the output `fi` object, `c`, gets assigned the same `fimath` object as inputs `a` and `b`.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The syntax `F.add(a,b)` is not supported. You must use the syntax `add(F,a,b)`.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide` | `sub` | `sum`

Topics

"fimath Rules for Fixed-Point Arithmetic"

assignmentquantizer

Package: embedded

Create `quantizer` object with `fi` object attributes

Syntax

```
q = assignmentquantizer(a)
```

Description

`q = assignmentquantizer(a)` creates a `quantizer` object `q` that is used in assignment operations for the `fi` object `a`. To use this object to quantize values, use `quantize`.

Examples

Create `quantizer` Object from `fi` Object

Use `assignmentquantizer` to create a `quantizer` object with the same quantization attributes as a `fi` object.

```
F = fimath('RoundingMethod','Convergent','OverflowAction','Saturate');
a = fi([],0,16,13,F);
q = assignmentquantizer(a)
```

```
q =


        DataMode = ufixed
       RoundMode = convergent
    OverflowMode = saturate
          Format = [16  13]
```

Input Arguments

a — Properties used for quantization
`fi` object

Properties used for quantization, specified as a `fi` object.

Data Types: `fi`

Version History

Introduced in R2008a

See Also

`quantize` | `quantizer` | `fi`

atan2

Four-quadrant inverse tangent of fixed-point values

Syntax

```
z = atan2(y,x)
```

Description

`z = atan2(y,x)` returns the four-quadrant arctangent of `fi` inputs `y` and `x`.

Examples

Calculate Arctangent of Fixed-Point Input Values

Use the `atan2` function to calculate the arctangent of unsigned and signed fixed-point input values.

Unsigned Input Values

This example uses unsigned, 16-bit word length values.

```
y = fi(0.125,0,16);
x = fi(0.5,0,16);
z = atan2(y,x)
```

```
z = 
    0.2450

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 15
```

Signed Input Values

This example uses signed, 16-bit word length values.

```
y = fi(-0.1,1,16);
x = fi(-0.9,1,16);
z = atan2(y,x)
```

```
z = 
   -3.0309

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
          WordLength: 16
      FractionLength: 13
```

Input Arguments

y — *y*-coordinates
scalar | vector | matrix | multidimensional array

y-coordinates, specified as a scalar, vector, matrix, or multidimensional array.

`y` and `x` can be real-valued, signed or unsigned scalars, vectors, matrices, or `N`-dimensional arrays containing fixed-point angle values in radians. The inputs `y` and `x` must be the same size. If they are not the same size, at least one input must be a scalar value. Valid data types of `y` and `x` are:

- `fi` single
- `fi` double
- `fi` fixed-point with binary point scaling
- `fi` scaled double with binary point scaling

Data Types: `fi`

x — *x*-coordinates
scalar | vector | matrix | multidimensional array

x-coordinates, specified as a scalar, vector, matrix, or multidimensional array.

`y` and `x` can be real-valued, signed or unsigned scalars, vectors, matrices, or `N`-dimensional arrays containing fixed-point angle values in radians. The inputs `y` and `x` must be the same size. If they are not the same size, at least one input must be a scalar value. Valid data types of `y` and `x` are:

- `fi` single
- `fi` double
- `fi` fixed-point with binary point scaling
- `fi` scaled double with binary point scaling

Data Types: `fi`

Output Arguments

z — Four-quadrant arctangent
scalar | vector | matrix | multidimensional array

Four-quadrant arctangent, returned as a scalar, vector, matrix, or multidimensional array.

`z` is the four-quadrant arctangent of `y` and `x`. The `numerictype` of `z` depends on the signedness of `y` and `x`:

- If either `y` or `x` is signed, then `z` is a signed, fixed-point number in the range [–pi,pi]. It has a 16-bit word length and 13-bit fraction length (`numerictype(1,16,13)`).
- If both `y` and `x` are unsigned, then `z` is an unsigned, fixed-point number in the range [0,pi/2]. It has a 16-bit word length and 15-bit fraction length (`numerictype(0,16,15)`).

The output, `z`, is always associated with the default `fimath`.

More About

Four-Quadrant Arctangent

The four-quadrant arctangent is defined as follows, with respect to the `atan` function:

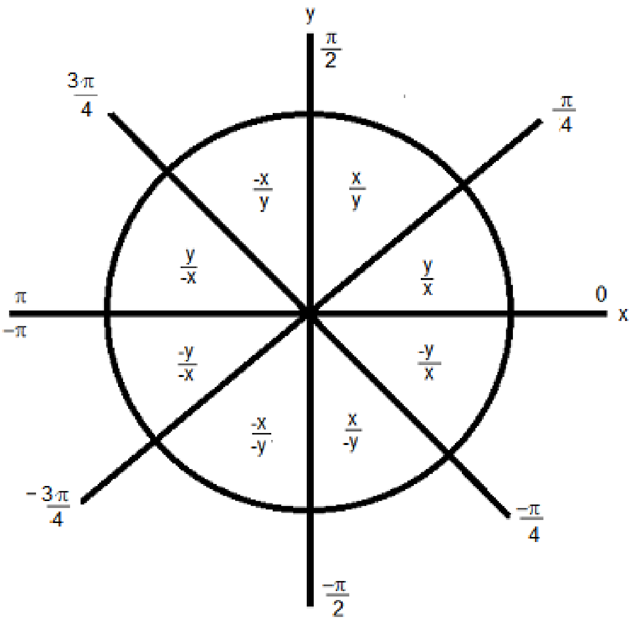
$$\operatorname{atan2}(y, x) = \begin{cases} \operatorname{atan}\left(\frac{y}{x}\right) & x > 0 \\ \pi + \operatorname{atan}\left(\frac{y}{x}\right) & y \geq 0, x < 0 \\ -\pi + \operatorname{atan}\left(\frac{y}{x}\right) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ 0 & y = 0, x = 0 \end{cases}$$

Algorithms

The `atan2` function computes the four-quadrant arctangent of fixed-point inputs using an 8-bit lookup table as follows:

1. Divide the input absolute values to get an unsigned, fractional, fixed-point, 16-bit ratio between 0 and 1. The absolute values of y and x determine which value is the divisor.

   The signs of the `y` and `x` inputs determine in what quadrant their ratio lies. The input with the larger absolute value is used as the denominator, thus producing a value between 0 and 1.



2. Compute the table index, based on the 16-bit, unsigned, stored integer value:
   - a. Use the 8 most-significant bits to obtain the first value from the table.
   - b. Use the next-greater table value as the second value.
3. Use the 8 least-significant bits to interpolate between the first and second values using nearest neighbor linear interpolation. This interpolation produces a value in the range [0, pi/4).
4. Perform octant correction on the resulting angle, based on the values of the original `y` and `x` inputs.

This arctangent calculation is accurate only to within the top 16 most-significant bits of the input.

fimath Propagation Rules

The `atan2` function ignores and discards any `fimath` attached to the inputs. The output, `z`, is always associated with the default `fimath`.

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`atan2` | `sin` | `angle` | `cos` | `cordicatan2`

autofixexp

Automatically change scaling of fixed-point data types

Syntax

```
autofixexp
```

Description

The `autofixexp` script automatically changes the scaling for model objects that specify fixed-point data types. However, if an object's **Lock output data type setting against changes by the fixed-point tools** parameter is selected, the script refrains from scaling that object.

This script collects range data for model objects, either from design minimum and maximum values that objects specify explicitly, or from logged minimum and maximum values that occur during simulation. Based on these values, the tool changes the scaling of fixed-point data types in a model so as to maximize precision and cover the range.

You can specify design minimum and maximum values for model objects using parameters typically titled **Output minimum** and **Output maximum**. See “Blocks That Allow Signal Range Specification” for a list of Simulink blocks that permit you to specify these values. In the autoscaling procedure that the `autofixexp` script executes, design minimum and maximum values take precedence over the simulation range.

If you intend to scale fixed-point data types using simulation minimum and maximum values, the script yields meaningful results when exercising the full range of values over which your design is meant to run. Therefore, the simulation you run prior to using `autofixexp` must simulate your design over its full intended operating range. It is especially important that you use simulation inputs with appropriate speed and amplitude profiles for dynamic systems. The response of a linear dynamic system is frequency dependent. For example, a bandpass filter will show almost no response to very slow and very fast sinusoid inputs, whereas the signal of a sinusoid input with a frequency in the passband will be passed or even significantly amplified. The response of nonlinear dynamic systems can have complicated dependence on both the signal speed and amplitude.

Note If you already know the simulation range you need to cover, you can use an alternate autoscaling technique described in the `fixptbestprec` reference page.

To control the parameters associated with automatic scaling, such as safety margins, use the Fixed-Point Tool.

To learn how to use the Fixed-Point Tool, refer to “Propose Fraction Lengths Using Simulation Range Data”.

Version History

Introduced before R2006a

See Also

`fxptdlg`

bin

Package: `embedded`

Unsigned binary representation of stored integer of `fi` object

Syntax

`b = bin(a)`

Description

`b = bin(a)` returns the stored integer of `fi` object `a` in unsigned binary format as a character vector.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

Tip `bin` returns the unsigned binary representation of the stored integer of a `fi` object. To obtain the binary representation of the real-world value of a `fi` object, use `dec2bin`.

Examples

View Stored Integer of `fi` Object in Unsigned Binary Format

Create a signed `fi` object with values `-1` and `1`, a word length of 8 bits, and a fraction length of 7 bits.

```
a = fi([-1 1], 1, 8, 7)
```

```
a =
   -1.0000    0.9922

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Find the unsigned binary representation of the stored integers of `fi` object `a`.

```
b = bin(a)
```

```
b =
'10000000   01111111'
```

Input Arguments

a — Input array
`fi` object

Input array, specified as a `fi` object.

Data Types: `fi`

Version History

Introduced before R2006a

See Also

`dec` | `hex` | `storedInteger` | `oct` | `dec2hex` | `dec2base` | `dec2bin`

bin2num

Convert two's complement binary string to number using `quantizer` object

Syntax

```
y = bin2num(q,b)
```

Description

`y = bin2num(q,b)` converts the binary character vector `b` to a numeric array `y` using the properties of the `quantizer` object `q`.

If `b` is a cell array containing binary strings, then `y` will be a cell array of the same dimension containing numeric arrays.

[`y1`,`y2`,...] = bin2num(`q`,`b1`,`b2`,...) converts the binary character vectors `b1`, `b2`, ... to numeric arrays `y1`, `y2`, ...

Examples

Convert Between Binary String and Numeric Array

Convert between a binary character vector and a numeric array using the properties specified in a `quantizer` object.

Convert Numeric Array to Binary String

Create a `quantizer` object specifying a word length of 4 bits and a fraction length of 3 bits. The other properties of the `quantizer` object take the default values of specifying a signed, fixed-point data type, rounding towards negative infinity, and saturate on overflow.

```
q = quantizer([4 3])
```

```
q = 


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [4  3]
```

Create an array of numeric values.

```
[a,b] = range(q);
x = (b:-eps(q):a)
```

```
x = 1×16

    0.8750    0.7500    0.6250    0.5000    0.3750    0.2500    0.1250         0   -0.1250   -0.2
```

Convert the numeric vector `x` to binary representation using the properties specified by the `quantizer` object `q`. Note that `num2bin` always returns the binary representations in a column.

```
b = num2bin(q,x)
```

```
b = 16x4 char array
    '0111'
    '0110'
    '0101'
    '0100'
    '0011'
    '0010'
    '0001'
    '0000'
    '1111'
    '1110'
    '1101'
    '1100'
    '1011'
    '1010'
    '1001'
    '1000'
```

Use `bin2num` to perform the inverse operation.

```
y = bin2num(q,b)
```

```
y = 16×1

    0.8750
    0.7500
    0.6250
    0.5000
    0.3750
    0.2500
    0.1250
         0
   -0.1250
   -0.2500
      ⋮
```

Convert Binary String to Numeric Array

All of the 3-bit fixed-point two's-complement numbers in fractional form are given by:

```
q = quantizer([3 2]);
b = ['011  111'
     '010  110'
     '001  101'
     '000  100'];
```

Use `bin2num` to view the numeric equivalents of these values.

```
x = bin2num(q,b)
```

```
x = 4×2
```

```
    0.7500   -0.2500
    0.5000   -0.5000
    0.2500   -0.7500
         0   -1.0000
```

Input Arguments

q — Data type properties to use for conversion
`quantizer` object

Data type properties to use for conversion, specified as a `quantizer` object.

Example: `q = quantizer([16 15]);`

b — Binary string to convert
character vector | character array | cell array

Binary string to convert, specified as a character vector, character array, or cell array containing binary strings.

Data Types: `string` | `char` | `cell`

Tips

- `bin2num` and `num2bin` are inverses of one another. Note that `num2bin` always returns the binary representations in a column.

Algorithms

- The fixed-point binary representation is two's complement.
- The floating-point binary representation is in IEEE Standard 754 style.
- If there are fewer binary digits than are necessary to represent the number, then fixed-point zero-pads on the left, and floating-point zero-pads on the right.

Version History

Introduced before R2006a

See Also

`num2bin` | `quantizer` | `hex2num` | `num2hex` | `num2int`

bitand

Bitwise AND of two fi objects

Syntax

```
c = bitand(a,b)
```

Description

c = bitand(a,b) returns the bitwise AND of fi objects a and b in fi object c.

The numerictype properties associated with a and b must be identical. If both inputs have a local fimath object, the fimath objects must be identical. If the numerictype is signed, then the bit representation of the stored integer is in two's complement representation.

a and b must have the same dimensions unless one is a scalar.

bitand only supports fi objects with fixed-point data types.

Examples

Compute Bitwise AND of Two fi Objects

Create a truth table for the logical AND operation.

```
A = fi([0 1; 0 1]);
B = fi([0 0; 1 1]);
TTable = bitand(A, B)
```

```
TTable =

     0     0
     0     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14
```

bitand returns 1 only if both bit-wise inputs are 1.

Input Arguments

a,b — Input values
scalars | vectors | matrices | multidimensional arrays

Input values, specified as scalars, vectors, matrices, or multidimensional arrays.a and b must have the same dimensions unless one is a scalar. Inputs a and b must be fi objects with fixed-point data types and identical numerictype properties. If both inputs have a local fimath object, the fimath objects must be identical.

Data Types: `fi`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Slope-bias scaled `fi` objects are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor`

bitandreduce

Reduce consecutive slice of bits to one bit by performing bitwise AND operation

Syntax

```
c = bitandreduce(a)
c = bitandreduce(a, lidx)
c = bitandreduce(a, lidx, ridx)
```

Description

`c = bitandreduce(a)` performs a bitwise AND operation on the entire set of bits in the fixed-point input, `a`, and returns the result as an unsigned integer of word length 1.

`c = bitandreduce(a, lidx)` performs a bitwise AND operation on a consecutive range of bits, starting at position `lidx` and ending at the LSB (the bit at position 1).

`c = bitandreduce(a, lidx, ridx)` performs a bitwise AND operation on a consecutive range of bits, starting at position `lidx` and ending at position `ridx`.

The `bitandreduce` arguments must satisfy the following condition:

```
a.WordLength >= lidx >= ridx >= 1
```

Examples

Perform Bitwise AND Operation on an Entire Set of Bits

Create a fixed-point number.

```
a = fi(73,0,8,0);
disp(bin(a))
```

```
01001001
```

Perform a bitwise AND operation on the entire set of bits in `a`.

```
c = bitandreduce(a)
```

```
c = 
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

Because the bits of `a` do not all have a value of 1, the output has a value of 0.

Perform Bitwise AND Operation on a Range of Bits in a Vector

Create a fixed-point vector.

```
a = fi([12, 4, 8, 15],0,8,0);
disp(bin(a))
```

```
00001100   00000100   00001000   00001111
```

Perform a bitwise AND operation on the bits of each element of `a`, starting at position `fi(4)`.

```
c = bitandreduce(a, fi(4))
```

```
c =
     0     0     0     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

The only element in output `c` with a value of 1 is the 4th element. This is because it is the only element of `a` that had only 1's between positions `fi(4)` and 1.

Perform Bitwise AND Operation on a Range of Bits in a Matrix

Create a fixed-point matrix.

```
a = fi([7, 8, 1; 5, 9, 5; 8, 37, 2], 0, 8, 0);
disp(bin(a))
```

```
00000111   00001000   00000001
00000101   00001001   00000101
00001000   00100101   00000010
```

Perform a bitwise AND operation on the bits of each element of matrix `a` beginning at position 3 and ending at position 1.

```
c = bitandreduce(a, 3, 1)
```

```
c =
     1     0     0
     0     0     0
     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

There is only one element in output `c` with a value of 1. This condition occurs because the corresponding element in `a` is the only element with only 1's between positions 3 and 1.

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects.

`bitandreduce` supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. `bitandreduce` performs the operation on a two's complement bit representation of the stored integer.

Data Types: fixed-point `fi`

`lidx` — Start position of range
scalar

Start position of range specified as a scalar of built-in type. `lidx` represents the position in the range closest to the MSB.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ridx` — End position of range
scalar

End position of range specified as a scalar of built-in type. `ridx` represents the position in the range closest to the LSB (the bit at position 1).

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

c — Output array
scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. `c` is unsigned with word length 1.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL®, generates the bitwise AND operator operating on a set of individual slices.

For Verilog®, generates the reduce operator:

```
&a[lidx:ridx]
```

See Also

`bitconcat` | `bitorreduce` | `bitsliceget` | `bitxorreduce`

bitcmp

Bitwise complement of `fi` object

Syntax

`c = bitcmp(a)`

Description

`c = bitcmp(a)` returns the bitwise complement of `fi` object `a`. If `a` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

`bitcmp` only supports `fi` objects with fixed-point data types. `a` can be a scalar `fi` object or a vector `fi` object.

Examples

This example shows how to get the bitwise complement of a `fi` object. Consider the following unsigned fixed-point `fi` object with a value of 10, word length 4, and fraction length 0:

```
a = fi(10,0,4,0);
disp(bin(a))
```

```
1010
```

Complement the values of the bits in `a`:

```
c = bitcmp(a);
disp(bin(c))
```

```
0101
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitand` | `bitget` | `bitor` | `bitset` | `bitxor`

bitconcat

Concatenate bits of `fi` objects

Syntax

```
y = bitconcat(a)
y = bitconcat(a, b, ...)
```

Description

`y = bitconcat(a)` concatenates the bits of the elements of fixed-point `fi` input array, `a`.

`y = bitconcat(a, b, ...)` concatenates the bits of the fixed–point `fi` inputs.

Examples

Concatenate the Elements of a Vector

Create a fixed-point vector.

```
a = fi([1,2,5,7],0,4,0);
disp(bin(a))
```

```
0001   0010   0101   0111
```

Concatenate the bits of the elements of `a`.

```
y = bitconcat(a)
```

```
y =
        4695

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 0
```

```
disp(bin(y))
```

```
0001001001010111
```

The word length of the output, `y`, equals the sum of the word lengths of each element of `a`.

Concatenate the Bits of Two fi Objects

Create two fixed-point numbers.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

```
b = fi(10,0,4,0);
disp(bin(b))
```

```
1010
```

Concatenate the bits of the two inputs.

```
y = bitconcat(a,b)
```

```
y =
    90

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

```
disp(bin(y))
```

```
01011010
```

The output, `y`, is unsigned with a word length equal to the sum of the word lengths of the two inputs, and a fraction length of 0.

Perform Element-by-Element Concatenation of Two Vectors

When `a` and `b` are both vectors of the same size, `bitconcat` performs element-wise concatenation of the two vectors and returns a vector.

Create two fixed-point vectors of the same size.

```
a = fi([1,2,5,7],0,4,0);
disp(bin(a))
```

```
0001   0010   0101   0111
```

```
b = fi([7,4,3,1],0,4,0);
disp(bin(b))
```

```
0111   0100   0011   0001
```

Concatenate the elements of `a` and `b`.

```
y = bitconcat(a,b)
```

```
y =
    23    36    83   113

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

```
disp(bin(y))
```

```
00010111   00100100   01010011   01110001
```

The output, y, is a vector of the same length as the input vectors, and with a word length equal to the sum of the word lengths of the two input vectors.

Perform Element-by-Element Concatenation of Two Matrices

When the inputs are both matrices of the same size, bitconcat performs element-wise concatenation of the two matrices and returns a matrix of the same size.

Create two fixed-point matrices.

```
a = fi([1,2,5;7,4,5;3,1,12],0,4,0);
disp(bin(a))
```

```
0001   0010   0101
0111   0100   0101
0011   0001   1100
```

```
b = fi([6,1,7;7,8,1;9,7,8],0,4,0);
disp(bin(b))
```

```
0110   0001   0111
0111   1000   0001
1001   0111   1000
```

Perform element-by-element concatenation of the bits of a and b.

```
y = bitconcat(a,b)
```

```
y = 
    22    33    87
   119    72    81
    57    23   200

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

```
disp(bin(y))
```

```
00010110   00100001   01010111
01110111   01001000   01010001
00111001   00010111   11001000
```

The output, y, is a matrix with word length equal to the sum of the word lengths of a and b.

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point fi objects. bitconcat accepts varargin number of inputs for concatenation.

Data Types: fixed-point fi

b — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `b` is nonscalar, it must have the same dimension as the other inputs.

Data Types: fixed-point `fi`

Output Arguments

y — Output array
scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of unsigned fixed-point `fi` objects.

The output array has word length equal to the sum of the word lengths of the inputs and a fraction length of zero. The bit representation of the stored integer is in two's complement representation. Scaling does not affect the result type and value.

If the inputs are all scalar, then `bitconcat` concatenates the bits of the inputs and returns a scalar.

If the inputs are all arrays of the same size, then `bitconcat` performs element-wise concatenation of the bits and returns an array of the same size.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL, generates the concatenation operator: `(a & b)`.

For Verilog, generates the concatenation operator: `{a , b}`.

See Also

`bitand` | `bitcmp` | `bitor` | `bitreplicate` | `bitget` | `bitset` | `bitsliceget` | `bitxor`

bitget

Get bits at certain positions

Syntax

```
c = bitget(a, bit)
```

Description

`c = bitget(a, bit)` returns the values of the bits at the positions specified by `bit` in `a` as unsigned integers of word length 1.

Examples

Get Bit When Input and Index Are Both Scalar

Consider the following unsigned fixed-point `fi` number with a value of 85, word length 8, and fraction length 0:

```
a = fi(85,0,8,0);
disp(bin(a))
```

```
01010101
```

Get the binary representation of the bit at position 4:

```
c = bitget(a,4);
```

`bitget` returns the bit at position 4 in the binary representation of `a`.

Get Bit When Input Is a Matrix and the Index Is a fi

Begin with a signed fixed-point 3-by-3 matrix with word length 4 and fraction length 0.

```
a = fi([2 3 4;6 8 2;3 5 1],0,4,0);
disp(bin(a))
```

```
0010   0011   0100
0110   1000   0010
0011   0101   0001
```

Get the binary representation of the bits at a specified position.

```
c = bitget(a,fi(2))
```

```
c =
     1     1     0
     1     0     1
     1     0     0
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

MATLAB® returns a matrix of the bits in position `fi(2)` of `a`. The output matrix has the same dimensions as `a`, and a word length of 1.

Get Bit When Both Input and Index Are Vectors

Begin with a signed fixed-point vector with word length 16, fraction length 4.

```
a = fi([86 6 53 8 1],0,16,4);
disp(bin(a))
```

```
0000010101100000   0000000001100000   0000001101010000   0000000010000000   0000000000010000
```

Create a vector that specifies the positions of the bits to get.

```
bit = [1,2,5,7,4]
```

```
bit = 1×5

     1     2     5     7     4
```

Get the binary representation of the bits of `a` at the positions specified in `bit`.

```
c = bitget(a,bit)
```

```
c = 
     0     0     1     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

`bitget` returns a vector of the bits of `a` at the positions specified in `bit`. The output vector has the same length as inputs, `a` and `bit`, and a word length of 1.

Get Bit When Input Is Scalar and Index Is a Vector

Create a default `fi` object with a value of `pi`.

```
a = fi(pi);
disp(bin(a))
```

```
0110010010001000
```

The default object is signed with a word length of 16.

Create a vector of the positions of the bits you want to get in `a`, and get the binary representation of those bits.

```
bit = fi([15,3,8,2]);
c = bitget(a,bit)
```

```
c =
     1     0     1     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

MATLAB® returns a vector of the bits in `a` at the positions specified by the index vector, `bit`.

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `a` and `bit` are both nonscalar, they must have the same dimension. If `a` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

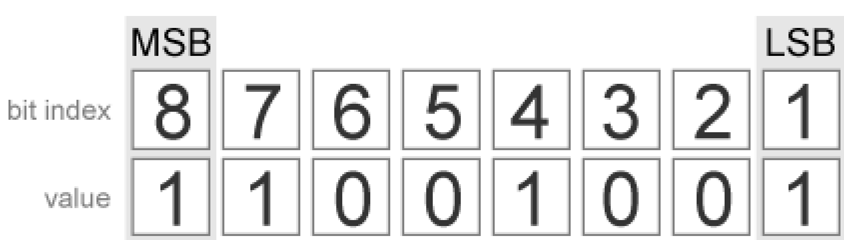
Data Types: fixed-point `fi`

bit — Bit index
scalar | vector | matrix | multidimensional array

Bit index, specified as a scalar, vector, matrix or multidimensional array of `fi` objects or built-in data types. If `a` and `bit` are both nonscalar, they must have the same dimension. `bit` must contain integer values between `1` and the word length of `a`, inclusive. The `LSB` (right-most bit) is specified by bit index `1` and the `MSB` (left-most bit) is specified by the word length of `a`. `bit` does not need to be a vector of sequential bit positions; it can also be a variable index value.

```
a = fi(pi,0,8);
a.bin
```

```
11001001
```



Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

c — Output array
scalar | vector | matrix | multidimensional array

Output array, specified as an unsigned scalar, vector, matrix, or multidimensional array with `WordLength` 1.

If `a` is an array and `bit` is a scalar, `c` is an unsigned array with word length 1. This unsigned array comprises the values of the bits at position `bit` in each fixed-point element in `a`.

If `a` is a scalar and `bit` is an array, `c` is an unsigned array with word length 1. This unsigned array comprises the values of the bits in `a` at the positions specified in `bit`.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL, generates the slice operator: `a(idx)`.

For Verilog, generates the slice operator: `a[idx]`.

See Also

`bitand` | `bitcmp` | `bitor` | `bitset` | `bitxor`

bitor

Bitwise OR of two fi objects

Syntax

```
c = bitor(a,b)
```

Description

`c = bitor(a,b)` returns the bitwise OR of fi objects a and b. The output is determined as follows:

- Elements in the output array c are assigned a value of 1 when the corresponding bit in either input array has a value of 1.
- Elements in the output array c are assigned a value of 0 when the corresponding bit in both input arrays has a value of 0.

The numerictype properties associated with a and b must be identical. If both inputs have a local fimath, their local fimath properties must be identical. If the numerictype is signed, then the bit representation of the stored integer is in two's complement representation.

a and b must have the same dimensions unless one is a scalar.

bitor only supports fi objects with fixed-point data types.

Examples

The following example finds the bitwise OR of fi objects *a* and *b*.

```
a = fi(-30,1,6,0);
b = fi(12, 1, 6, 0);
c = bitor(a,b)

c =

   -18

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

You can verify the result by examining the binary representations of *a*, *b* and *c*.

```
binary_a = a.bin
binary_b = b.bin
binary_c = c.bin

binary_a =

100010
```

```
binary_b =

001100

binary_c =

101110
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Slope-bias scaled `fi` objects are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitand` | `bitcmp` | `bitget` | `bitset` | `bitxor`

bitorreduce

Reduce consecutive slice of bits to one bit by performing bitwise OR operation

Syntax

```
c = bitorreduce(a)
c = bitorreduce(a, lidx)
c = bitorreduce(a, lidx, ridx)
```

Description

`c = bitorreduce(a)` performs a bitwise OR operation on the entire set of bits in the fixed-point input, `a`, and returns the result as an unsigned integer of word length 1.

`c = bitorreduce(a, lidx)` performs a bitwise OR operation on a consecutive range of bits, starting at position `lidx` and ending at the LSB (the bit at position 1).

`c = bitorreduce(a, lidx, ridx)` performs a bitwise OR operation on a consecutive range of bits, starting at position `lidx` and ending at position `ridx`.

The `bitorreduce` arguments must satisfy the following condition:

`a.WordLength >= lidx >= ridx >= 1`

Examples

Perform Bitwise OR Operation on an Entire Set of Bits

Create a fixed-point number.

```
a = fi(73,0,8,0);
disp(bin(a))
```

```
01001001
```

Perform a bitwise OR operation on the entire set of bits in `a`.

```
c = bitorreduce(a)
```

```
c = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

Because there is at least one bit in `a` with a value of 1, the output has a value of `1`.

Perform Bitwise OR Operation on a Range of Bits in a Vector

Create a fixed-point vector.

```
a=fi([12,4,8,15],0,8,0);
disp(bin(a))
```

```
00001100   00000100   00001000   00001111
```

Perform a bitwise OR operation on the bits of each element of `a`, starting at position `fi(4)`.

```
c=bitorreduce(a,fi(4))
```

```
c =
     1     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

All of the entries of output `c` have a value of `1` because all of the entries of `a` have at least one bit with a value of `1` between the positions `fi(4)` and 1.

Perform Bitwise OR Operation on a Range of Bits in a Matrix

Create a fixed-point matrix.

```
a = fi([7,8,1;5,9,5;8,37,2],0,8,0);
disp(bin(a))
```

```
00000111   00001000   00000001
00000101   00001001   00000101
00001000   00100101   00000010
```

Perform a bitwise OR operation on the bits of each element of matrix `a` beginning at position 5, and ending at position 2.

```
c = bitorreduce(a,5,2)
```

```
c =
     1     1     0
     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

There is only one element in output `c` that does not have a value of `1`. This condition occurs because the corresponding element in `a` is the only element of `a` that does not have any bits with a value of `1` between positions 5 and 2.

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects.

`bitorreduce` supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. `bitorreduce` performs the operation on a two's complement bit representation of the stored integer.

Data Types: fixed-point `fi`

`lidx` — Start position of range
scalar

Start position of range specified as a scalar of built-in type. `lidx` represents the position in the range closest to the MSB.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ridx` — End position of range
scalar

End position of range specified as a scalar of built-in type. `ridx` represents the position in the range closest to the LSB (the bit at position 1).

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

c — Output array
scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. `c` is unsigned with word length 1.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL, generates the bitwise OR operator operating on a set of individual slices.

For Verilog, generates the reduce operator:

```
|a[lidx:ridx]
```

See Also

`bitandreduce` | `bitconcat` | `bitsliceget` | `bitxorreduce`

bitreplicate

Replicate and concatenate bits of `fi` object

Syntax

c = `bitreplicate(`*a*`,`*n*`)`

Description

c = `bitreplicate(`*a*`,`*n*`)` concatenates the bits in `fi` object *a* *n* times and returns an unsigned fixed-point value. The word length of the output `fi` object *c* is equal to *n* times the word length of *a* and the fraction length of *c* is zero. The bit representation of the stored integer is in two's complement representation.

The input `fi` object can be signed or unsigned. `bitreplicate` concatenates signed and unsigned bits the same way.

`bitreplicate` only supports `fi` objects with fixed-point data types.

`bitreplicate` does not support inputs with complex data types.

Sign and scaling of the input `fi` object does not affect the result type and value.

Examples

The following example uses `bitreplicate` to replicate and concatenate the bits of `fi` object `a`.

```
a = fi(14,0,6,0);
a_binary = a.bin
c = bitreplicate(a,2);
c_binary = c.bin
```

MATLAB returns the following:

```
a_binary =

001110


c_binary =

001110001110
```

Version History

Introduced in R2008a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`bitand` | `bitconcat` | `bitget` | `bitset` | `bitor` | `bitsliceget` | `bitxor`

bitrol

Bitwise rotate left

Syntax

```
c = bitrol(a, k)
```

Description

`c = bitrol(a, k)` returns the value of the fixed-point `fi` object, `a`, rotated left by `k` bits. `bitrol` rotates bits from the most significant bit (MSB) side into the least significant bit (LSB) side. It performs the rotate left operation on the stored integer bits of `a`.

`bitrol` does not check overflow or underflow. It ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

`a` and `c` have the same `fimath` and `numerictype` properties.

Examples

Rotate the Bits of a fi Object Left

Create an unsigned fixed-point `fi` object with a value of 10, word length 4, and fraction length 0.

```
a = fi(10,0,4,0);
disp(bin(a))
```

```
1010
```

Rotate `a` left 1 bit.

```
disp(bin(bitrol(a,1)))
```

```
0101
```

Rotate `a` left 2 bits.

```
disp(bin(bitrol(a,2)))
```

```
1010
```

Rotate Bits in a Vector Left

Create a vector of `fi` objects.

```
a = fi([1,2,5,7],0,4,0)
```

```
a =
     1     2     5     7
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 4
        FractionLength: 0
```

```
disp(bin(a))
```

```
0001   0010   0101   0111
```

Rotate the bits in vector `a` left 1 bit.

```
disp(bin(bitrol(a,1)))
```

```
0010   0100   1010   1110
```

Rotate Bits Left Using fi to Specify Number of Bits to Rotate

Create an unsigned fixed-point `fi` object with a value 10, word length 4, and fraction length 0.

```
a = fi(10,0,4,0);
```

```
disp(bin(a))
```

```
1010
```

Rotate `a` left 1 bit where `k` is a `fi` object.

```
disp(bin(bitrol(a,fi(1))))
```

```
0101
```

Input Arguments

a — Data that you want to rotate
scalar | vector | matrix | multidimensional array

Data that you want to rotate, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects. `a` can be signed or unsigned.

Data Types: fixed-point `fi`

Complex Number Support: Yes

k — Number of bits to rotate
non-negative, integer-valued scalar

Number of bits to rotate, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type. `k` can be greater than the word length of `a`. This value is always normalized to `mod(a.WordLength,k)`.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL, generates the `rol` operator.

For Verilog, generates the following expression (where `wl` is the word length of `a`:

```
a << idx || a >> wl - idx
```

See Also

`bitconcat` | `bitror` | `bitshift` | `bitsliceget` | `bitsll` | `bitsra` | `bitsrl`

bitror

Bitwise rotate right

Syntax

```
c = bitror(a, k)
```

Description

`c = bitror(a, k)` returns the value of the fixed-point `fi` object, `a`, rotated right by `k` bits. `bitror` rotates bits from the least significant bit (LSB) side into the most significant bit (MSB) side. It performs the rotate right operation on the stored integer bits of `a`.

`bitror` does not check overflow or underflow. It ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

`a` and `c` have the same `fimath` and `numerictype` properties.

Examples

Rotate Bits of a fi Object Right

Create an unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

Rotate `a` right 1 bit.

```
disp(bin(bitror(a,1)))
```

```
1010
```

Rotate `a` right 2 bits.

```
disp(bin(bitror(a,2)))
```

```
0101
```

Rotate Bits in a Vector Right

Create a vector of `fi` objects.

```
a = fi([1,2,5,7],0,4,0);
disp(bin(a))
```

```
0001   0010   0101   0111
```

Rotate the bits in vector `a` right 1 bit.

```
disp(bin(bitror(a,fi(1))))
```

```
1000   0001   1010   1011
```

Rotate Bits Right Using fi to Specify Number of Bits to Rotate

Create an unsigned fixed-point `fi` object with a value 5, word length 4, and fraction length 0.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

Rotate `a` right 1 bit where `k` is a `fi` object.

```
disp(bin(bitror(a,fi(1))))
```

```
1010
```

Input Arguments

a — Data that you want to rotate
scalar | vector | matrix | multidimensional array

Data that you want to rotate, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects. `a` can be signed or unsigned.

Data Types: fixed-point `fi`

Complex Number Support: Yes

k — Number of bits to rotate
non-negative, integer-valued scalar

Number of bits to rotate, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type. `k` can be greater than the word length of `a`. This value is always normalized to `mod(a.WordLength,k)`.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL, generates the `ror` operator.

For Verilog, generates the following expression (where `wl` is the word length of `a`:

```
a >> idx || a << wl - idx
```

See Also

`bitrol` | `bitconcat` | `bitshift` | `bitsliceget` | `bitsll` | `bitsra` | `bitsrl`

bitset

Package: embedded

Set bit at specific location

Syntax

```
C = bitset(A,bit)
C = bitset(A,bit,V)
```

Description

`C = bitset(A,bit)` returns the value of `A` with position `bit` set to `1` (on).

`C = bitset(A,bit,V)` returns the value of `A` with position `bit` set to `V`.

Examples

Set Bit at Certain Position

Begin with an unsigned fixed-point `fi` number with a value of 5, word length 4, and fraction length 0.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

Set the bit at position 4 to `1` (on).

```
c = bitset(a,4);
disp(bin(c))
```

```
1101
```

Set Bit at Certain Position in Vector

Consider the following fixed-point vector with word length 4 and fraction length 0.

```
a = fi([0 1 8 2 4],0,4,0);
disp(bin(a))
```

```
0000   0001   1000   0010   0100
```

In each element of vector `a`, set the bits at position 2 to `1`.

```
c = bitset(a,2,1);
disp(bin(c))
```

```
0010   0011   1010   0010   0110
```

Set Bit at Certain Position with Fixed Point Index

Consider the following fixed-point scalar with a value of 5.

```
a = fi(5,0,4,0);
disp(bin(a))
```

```
0101
```

Set the bit at position `fi(2)` to `1`.

```
c = bitset(a,fi(2),1);
disp(bin(c))
```

```
0111
```

Set Bit When Index Is Vector

Create a `fi` object with a value of `pi`.

```
a = fi(pi);
disp(bin(a))
```

```
0110010010001000
```

In this case, `a` is signed with a word length of 16.

Create a vector of the bit positions in `a` that you want to set to on. Then, get the binary representation of the resulting `fi` vector.

```
bit = fi([15,3,8,2]);
c = bitset(a,bit);
disp(bin(c))
```

```
0110010010001000   0110010010001100   0110010010001000   0110010010001010
```

Input Arguments

A — Input values
scalar | vector | matrix | multidimensional array

Input values, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If any of `A`, `bit`, or `V` are nonscalar, the other inputs must be scalar or arrays of the same size. If `A` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

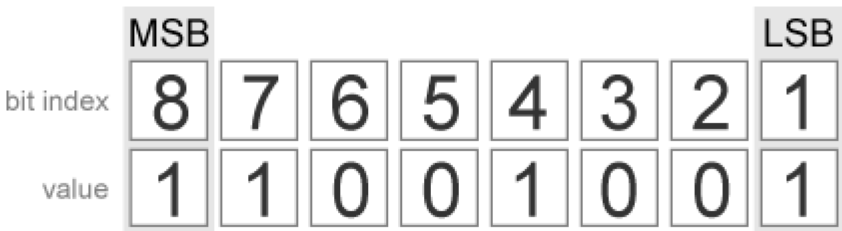
Data Types: `fi`
Complex Number Support: Yes

bit — Bit position
integer | integer array

Bit position, specified as an integer or integer array of `fi` objects or built-in data types. If any of `A`, `bit`, or `V` are nonscalar, the other inputs must be scalar or arrays of the same size. The values of `bit` must be between `1` and the word length of `A`, inclusive. The `LSB`, the right-most bit, is specified by bit index `1`. The `MSB`, the left-most bit, is specified by the word length of `A`.

```
a = fi(pi,0,8);
a.bin
```

```
ans =

    '11001001'
```



Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`V` — Bit value
scalar | vector | matrix | multidimensional array

Bit value of `A` at index `bit`, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in data types. If any of `A`, `bit`, or `V` are nonscalar, the other inputs must be scalar or arrays of the same size. `V` can have values of `0` or `1`. Any value other than `0` is automatically set to `1`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Output Arguments

`C` — Output array
scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects.

- If `A`, `bit`, and `V` are all scalars, then `C` is also a scalar.
- If any of `A`, `bit`, or `V` is an array, then `C` is the same size as that array.

Version History

Introduced before R2006a

R2022a: Scalar expansion support for `fi` `bitset`
Behavior changed in R2022a

Prior to R2022a, `fi` `bitset` required that the second and third input arguments be the same size, otherwise an error would occur.

```
A = fi(pi);
disp(bin(A))

bit = fi([15,3,8,2]);
C = bitset(A,bit,1);
disp(bin(C))
```

```
0110010010001000
The Second and third arguments to BITSET must be the same size.
```

Starting in R2022a, the input arguments `A`, `bit`, and `V` support scalar expansion. That is, if any of `A`, `bit`, or `V` are nonscalar, the other inputs can be scalar or arrays of the same size.

```
A = fi(pi);
disp(bin(A))

bit = fi([15,3,8,2]);
C = bitset(A,bit,1);
disp(bin(C))
```

```
0110010010001000
0110010010001000   0110010010001100   0110010010001000   0110010010001010
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`bitand` | `bitcmp` | `bitget` | `bitor` | `bitxor`

bitshift

Shift bits specified number of places

Syntax

```
c = bitshift(a,k)
```

Description

`c = bitshift(a,k)` returns the value of `fi` object `a` shifted by `k` bits.

The shift is arithmetic and behaves like `b = a.*2^k` with the value of `b` cast to the type of input `a`. The cast of `b` may involve overflow or loss of precision.

The `OverflowAction` property of `a` is obeyed, but the `RoundingMethod` is always `Floor`. If obeying the `RoundingMethod` property of `a` is important, try using the `pow2` function.

When the overflow action of `a` is `Saturate`, the sign bit is always preserved. When the overflow action of `a` is `Wrap` and `k` is negative, the sign bit is preserved. When the overflow action of `a` is `Wrap` and `k` is positive, the sign bit may change.

- When `k` is positive, 0-valued bits are shifted in on the right.
- When `k` is negative and `a` is unsigned, or a signed and positive `fi` object, 0-valued bits are shifted in on the left.
- When `k` is negative and `a` is a signed and negative `fi` object, 1-valued bits are shifted in on the left.

Examples

Use `OverflowAction` Settings to Change Results of `bitshift`

This example highlights how changing the `OverflowAction` property of the `fimath` object can change the results returned by the `bitshift` function. Consider the following signed fixed-point `fi` object with a value of 3, word length 16, and fraction length 0.

```
a = fi(3,1,16,0);
```

By default, the `OverflowAction fimath` property is `Saturate`. When `a` is shifted such that it overflows, it is saturated to the maximum possible value.

```
for k=0:16
    b=bitshift(a,k);
    disp([num2str(k,'%02d'),'. ',bin(b)]);
end
```

```
00. 0000000000000011
01. 0000000000000110
02. 0000000000001100
03. 0000000000011000
```

```
04. 0000000000110000
05. 0000000001100000
06. 0000000011000000
07. 0000000110000000
08. 0000001100000000
09. 0000011000000000
10. 0000110000000000
11. 0001100000000000
12. 0011000000000000
13. 0110000000000000
14. 0111111111111111
15. 0111111111111111
16. 0111111111111111
```

Now change `OverflowAction` to `Wrap`. In this case, most significant bits shift off the "top" of `a` until the value is zero.

```
a = fi(3,1,16,0,'OverflowAction','Wrap');
for k=0:16
    b=bitshift(a,k);
    disp([num2str(k,'%02d'),'. ',bin(b)]);
end
```

```
00. 0000000000000011
01. 0000000000000110
02. 0000000000001100
03. 0000000000011000
04. 0000000000110000
05. 0000000001100000
06. 0000000011000000
07. 0000000110000000
08. 0000001100000000
09. 0000011000000000
10. 0000110000000000
11. 0001100000000000
12. 0011000000000000
13. 0110000000000000
14. 1100000000000000
15. 1000000000000000
16. 0000000000000000
```

Input Arguments

a — Input `fi` object

scalar | vector

Input `fi` object, specified as a scalar or vector. `a` can be any fixed-point numeric type.

Data Types: `fi`

k — Number of bits to shift by

scalar

Number of bits to shift by, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

c — Result of shifting a by k bits
fi object

Result of shifting `a` by `k` bits, returned as a `fi` object. The output `fi` object `c` has the same `numerictype` as `a`.

Version History
Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For efficient HDL code generation, use `bitsll`, `bitsrl`, or `bitsra` instead of `bitshift`.

See Also
`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitsll` | `bitsra` | `bitsrl` | `bitxor` | `pow2`

bitsliceget

Get consecutive slice of bits

Syntax

```
c = bitsliceget(a)
c = bitsliceget(a, lidx)
c = bitsliceget(a, lidx, ridx)
```

Description

`c = bitsliceget(a)` returns the entire set of bits in the fixed-point input `a`.

`c = bitsliceget(a, lidx)` returns a consecutive slice of bits from `a`, starting at position `lidx` and ending at the LSB (the bit at position 1).

`c = bitsliceget(a, lidx, ridx)` returns a consecutive slice of bits from `a`, starting at position `lidx` and ending at position `ridx`.

The `bitsliceget` arguments must satisfy the following condition:

`a.WordLength >= lidx >= ridx >= 1`

Examples

Get Entire Set of Bits

Begin with the following fixed-point number.

```
a = fi(85,0,8,0);
disp(bin(a))
```

```
01010101
```

Get the entire set of bits of `a`.

```
c = bitsliceget(a);
disp(bin(c))
```

```
01010101
```

Get a Slice of Consecutive Bits with Unspecified Endpoint

Begin with the following fixed-point number.

```
a = fi(85,0,8,0);
disp(bin(a))
```

```
01010101
```

Get the binary representation of the consecutive bits, starting at position 6.

```
c = bitsliceget(a,6);
disp(bin(c))
```

```
010101
```

Get a Slice of Consecutive Bits with Fixed-Point Indexes

Begin with the following fixed-point number.

```
a = fi(85,0,8,0);
disp(bin(a))
```

```
01010101
```

Get the binary representation of the consecutive bits from `fi(6)` to `fi(2)`.

```
c = bitsliceget(a,fi(6),fi(2));
disp(bin(c))
```

```
01010
```

Get a Specified Set of Consecutive Bits from Each Element of a Matrix

Begin with the following unsigned fixed-point 3-by-3 matrix.

```
a = fi([2 3 4;6 8 2;3 5 1],0,4,0);
disp(bin(a))
```

```
0010   0011   0100
0110   1000   0010
0011   0101   0001
```

Get the binary representation of a consecutive set of bits of matrix `a`. For each element, start at position 4 and end at position 2.

```
c = bitsliceget(a,4,2);
disp(bin(c))
```

```
001   001   010
011   100   001
001   010   000
```

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. If `a` has a signed `numerictype`, the bit representation of the stored integer is in two's complement representation.

Data Types: fixed-point `fi`

`lidx` — Start position for slice
scalar

Start position of slice specified as a scalar of built-in type. `lidx` represents the position in the slice closest to the MSB.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ridx` — End position for slice
scalar

End position of slice specified as a scalar of built-in type. `ridx` represents the position in the slice closest to the LSB (the bit at position 1).

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`c` — Output array
scalar | vector | matrix | multidimensional array

Fixed-point `fi` output, specified as a scalar, vector, matrix, or multidimensional array with no scaling. The word length is equal to slice length, `lidx-ridx+1`.

If `lidx` and `ridx` are equal, `bitsliceget` only slices one bit, and `bitsliceget(a, lidx, ridx)` is the same as `bitget(a, lidx)`.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset` | `bitxor`

bitsll

Bit shift left logical

Syntax

`c = bitsll(a, k)`

Description

`c = bitsll(a, k)` returns the result of a logical left shift by `k` bits on input `a` for fixed-point operations. `bitsll` shifts zeros into the positions of bits that it shifts left. The function does not check overflow or underflow. For floating-point operations, `bitsll` performs a multiply by 2^k.

`bitsll` ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fimath` and `numerictype` objects.

Examples

Shift Left a Signed fi Input

Shift a signed `fi` input left by 1 bit.

Create a `fi` object, and display its binary value.

```
a = fi(10,0,4,0);
disp(bin(a))
```

```
1010
```

Shift `a` left by 1 bit, and display its binary value.

```
disp(bin(bitsll(a,1)))
```

```
0100
```

Shift `a` left by 1 more bit.

```
disp(bin(bitsll(a,2)))
```

```
1000
```

Shift Left Using a fi Shift Value

Shift left a built-in `int8` input using a `fi` shift value.

```
k = fi(2);
a = int8(16);
bitsll(a,k)
```

```
ans = int8
    64
```

Shift Left a Built-in int8 Input

Use `bitsll` to shift an `int8` input left by 2 bits.

```
a = int8(4);
bitsll(a,2)
```

```
ans = int8
    16
```

Shift Left a Floating-Point Input

Scale a floating-point `double` input by 2^3.

```
a = double(16);
bitsll(a,3)
```

```
ans = 128
```

Input Arguments

a — Data that you want to shift
scalar | vector | matrix | multidimensional array

Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array of fi objects or built-in numeric types.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

k — Number of bits to shift
non-negative integer-valued scalar

Number of bits to shift, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Generates `sll` operator in VHDL code.

Generates << operator in Verilog code.

See Also
`bitsrl` | `bitsra` | `bitshift` | `pow2` | `bitconcat` | `bitrol` | `bitror`

bitsra

Bit shift right arithmetic

Syntax

`c=bitsra(a,k)`

Description

`c=bitsra(a,k)` returns the result of an arithmetic right shift by `k` bits on input `a` for fixed-point operations. For floating-point operations, it performs a multiply by 2^{-k}.

If the input is unsigned, `bitsra` shifts zeros into the positions of bits that it shifts right. If the input is signed, `bitsra` shifts the most significant bit (MSB) into the positions of bits that it shifts right.

`bitsra` ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fimath` and `numerictype` objects.

Examples

Shift Right a Signed fi Input

Create a signed fixed-point `fi` object with a value of –8, word length 4, and fraction length 0. Then display the binary value of the object.

```
a = fi(-8,1,4,0);
disp(bin(a))
```

```
1000
```

Shift `a` right by 1 bit.

```
disp(bin(bitsra(a,1)))
```

```
1100
```

`bitsra` shifts the MSB into the position of the bit that it shifts right.

Shift Right a Built-in int8 Input

Use `bitsra` to shift an `int8` input right by 2 bits.

```
a = int8(64);
bitsra(a,2)
```

```
ans = int8
    16
```

Shift Right Using a fi Shift Value

Shift right a built-in `int8` input using a `fi` shift value.

```
k = fi(2);
a = int8(64);
bitsra(a,k)
```

```
ans = int8
    16
```

Shift Right a Floating-Point Input

Scale a floating-point `double` input by 2^{-3}.

```
a = double(128);
bitsra(a,3)
```

```
ans = 16
```

Input Arguments

a — Data that you want to shift
scalar | vector | matrix | multidimensional array

Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in numeric types.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

k — Number of bits to shift
non-negative integer-valued scalar

Number of bits to shift, specified as a non-negative integer-valued scalar `fi` object or built-in numeric type.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Generates `sra` operator in VHDL code.

Generates >>> operator in Verilog code.

See Also
`bitsll` | `bitsrl` | `bitshift` | `pow2`

bitsrl

Bit shift right logical

Syntax

```
c = bitsrl(a, k)
```

Description

`c = bitsrl(a, k)` returns the result of a logical right shift by `k` bits on input `a` for fixed-point operations. `bitsrl` shifts zeros into the positions of bits that it shifts right. It does not check overflow or underflow.

`bitsrl` ignores `fimath` properties such as `RoundingMode` and `OverflowAction`.

When `a` is a `fi` object, `a` and `c` have the same associated `fimath` and `numerictype` objects.

Examples

Shift right a signed fi input

Shift a signed `fi` input right by 1 bit.

Create a signed fixed-point `fi` object with a value of -8, word length 4, and fraction length 0 and display its binary value.

```
a = fi(-8,1,4,0);
disp(bin(a))
```

```
1000
```

Shift `a` right by 1 bit, and display the binary value.

```
disp(bin(bitsrl(a,1)))
```

```
0100
```

`bitsrl` shifts a zero into the position of the bit that it shifts right.

Shift right using a fi shift value

Shift right a built-in `int8` input using a `fi` shift value.

```
k = fi(2);
a = int8(64);
bitsrl(a,k)
```

```
ans = int8
    16
```

Shift right a built-in uint8 input

Use `bitsrl` to shift a uint8 input right by 2 bits.

```
a = uint8(64);
bitsrl(a,2)
```

```
ans = uint8
    16
```

Input Arguments

a — Data that you want to shift
scalar | vector | matrix | multidimensional array

Data that you want to shift, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

k — Number of bits to shift
non-negative integer-valued scalar

Number of bits to shift, specified as a non-negative integer-valued scalar.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History
Introduced in R2007b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- Generated code might not handle out of range shifting.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Generates `srl` operator in VHDL.

Generates >> operator in Verilog.

See Also

`bitconcat` | `bitrol` | `bitror` | `bitshift` | `bitsliceget` | `bitsll` | `bitsra` | `pow2`

bitxor

Package: `embedded`

Bitwise XOR of two `fi` objects

Syntax

```
c = bitxor(a,b)
```

Description

`c = bitxor(a,b)` returns the bitwise exclusive OR of `fi` objects `a` and `b` in `fi` object `c`.

The output is determined as follows:

- Elements in the output array `c` are assigned a value of `1` when exactly one of the corresponding bits in the input arrays has a value of `1`.
- Elements in the output array `c` are assigned a value of `0` when the corresponding bits in the input arrays have the same value (e.g. both `1`'s or both `0`'s).

Note This function only supports `fi` objects with fixed-point data types. To compute bitwise XOR of other data types, use the `bitxor` function.

Examples

Bitwise Exclusive OR of Two `fi` Objects

Find the bitwise exclusive OR of `fi` objects `a` and `b`.

```
a = fi(-28,1,6,0);
b = fi(12, 1, 6, 0);
c = bitxor(a,b)
```

```
c = 
   -24

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

To verify the result, examine the binary representations of `a`, `b`, and `c`.

```
binary_a = a.bin
```

```
binary_a = 
'100100'
```

```
binary_b = b.bin
```

```
binary_b =
'001100'

binary_c = c.bin

binary_c =
'101000'
```

Input Arguments

`a`,`b` — Input fixed-point `fi` objects
scalars | vectors | matrices | multidimensional arrays

Input fixed-point `fi` objects, specified as scalars, vectors, matrices, or multidimensional arrays. `a` and `b` must have the same dimensions unless one is a scalar.

The `numerictype` properties associated with `a` and `b` must be identical. If both inputs have an attached `fimath` object, their local `fimath` properties must be identical. If the `numerictype` is signed, then the bit representation of the stored integer is in “Two's Complement” representation.

Data Types: `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Slope-bias scaled `fi` objects are not supported.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitand` | `bitcmp` | `bitget` | `bitor` | `bitset`

bitxorreduce

Reduce consecutive slice of bits to one bit by performing bitwise exclusive OR operation

Syntax

```
c = bitxorreduce(a)
c = bitxorreduce(a, lidx)
c = bitxorreduce(a, lidx, ridx)
```

Description

`c = bitxorreduce(a)` performs a bitwise exclusive OR operation on the entire set of bits in the fixed-point input, `a`. It returns the result as an unsigned integer of word length 1.

`c = bitxorreduce(a, lidx)` performs a bitwise exclusive OR operation on a consecutive range of bits. This operation starts at position `lidx` and ends at the LSB (the bit at position 1).

`c = bitxorreduce(a, lidx, ridx)` performs a bitwise exclusive OR operation on a consecutive range of bits, starting at position `lidx` and ending at position `ridx`.

The `bitxorreduce` arguments must satisfy the following condition:

```
a.WordLength >= lidx >= ridx >= 1
```

Examples

Perform Bitwise Exclusive OR Operation on an Entire Set of Bits

Create a fixed-point number.

```
a = fi(73,0,8,0);
disp(bin(a))
```

```
01001001
```

Perform a bitwise exclusive OR operation on the entire set of bits in `a`.

```
c = bitxorreduce(a)
```

```
c =
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

Perform Bitwise Exclusive OR Operation on a Range of Bits in a Vector

Create a fixed-point vector.

```
a = fi([12,4,8,15],0,8,0);
disp(bin(a))
```

```
00001100   00000100   00001000   00001111
```

Perform a bitwise exclusive OR operation on the bits of each element of `a`, starting at position `fi(4)`.

```
c = bitxorreduce(a,fi(4))
```

```
c =
     0     1     1     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

Perform a Bitwise Exclusive OR Operation on a Range of Bits in a Matrix

Create a fixed-point matrix.

```
a = fi([7,8,1;5,9,5;8,37,2],0,8,0);
disp(bin(a))
```

```
00000111   00001000   00000001
00000101   00001001   00000101
00001000   00100101   00000010
```

Perform a bitwise exclusive OR operation on the bits of each element of matrix `a` beginning at position 5 and ending at position 2.

```
c = bitxorreduce(a,5,2)
```

```
c =
     0     1     0
     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects.

`bitxorreduce` supports both signed and unsigned inputs with arbitrary scaling. The sign and scaling properties do not affect the result type and value. `bitxorreduce` performs the operation on a two's complement bit representation of the stored integer.

Data Types: fixed-point `fi`

`lidx` — Start position of range
scalar

Start position of range specified as a scalar of built-in type. `lidx` represents the position in the range closest to the MSB.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ridx` — End position of range
scalar

End position of range specified as a scalar of built-in type. `ridx` represents the position in the range closest to the LSB (the bit at position 1).

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`c` — Output array
scalar | vector | matrix | multidimensional array

Output array, specified as a scalar, vector, matrix, or multidimensional array of fixed-point `fi` objects. `c` is unsigned with word length 1.

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For VHDL, generates a set of individual slices.

For Verilog, generates the reduce operator:

```
^a[lidx:ridx]
```

See Also

`bitandreduce` | `bitconcat` | `bitorreduce` | `bitsliceget`

buildInstrumentedMex

Generate compiled C code function including logging instrumentation

Syntax

```
buildInstrumentedMex fcn -options
buildInstrumentedMex fcn_1... fcn_n -options -coder
```

Description

`buildInstrumentedMex fcn -options` translates the MATLAB file *`fcn`*`.m` to a MEX function and enables instrumentation for logging minimum and maximum values of all named and intermediate variables. Optionally, you can enable instrumentation for log2 histograms of all named, intermediate and expression values. The general syntax and options of `buildInstrumentedMex` and `fiaccel` are the same, except `buildIntstrumentedMex` has no `fi` object restrictions and supports the `'-coder'` option.

Note Like the `fiaccel` function, the `buildInstrumentedMex` function generates a MEX function. To generate C code, use the `codegen` function.

`buildInstrumentedMex fcn_1... fcn_n -options -coder` translates the MATLAB functions *`fcn_1`* through *`fcn_n`* to a MEX function and enables instrumentation for logging minimum and maximum values of all named and intermediate variables. Generating a MEX function for multiple entry-point functions requires the `'-coder'` option.

Note Generating a MEX function for multiple entry-point functions using the `buildInstrumentedMex` function requires a MATLAB Coder license.

Examples

Create Instrumented MEX Function

This example shows how to create an instrumented MEX function, run a test bench, then view logged results.

Define prototype input arguments.

```
n = 128;
x = complex(zeros(n,1));
w = fi_radix2twiddles(n);
```

Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name. Use the `-histogram` option to compute histograms.

If you have a MATLAB® Coder™ license, you can also add the `-coder` option. In this case,

```
buildInstrumentedMex testfft -o testfft_instrumented -args {x,coder.Constant(w)} -histogram
```

If you have a MATLAB® Coder™ license, you can also add the -coder option. For example,

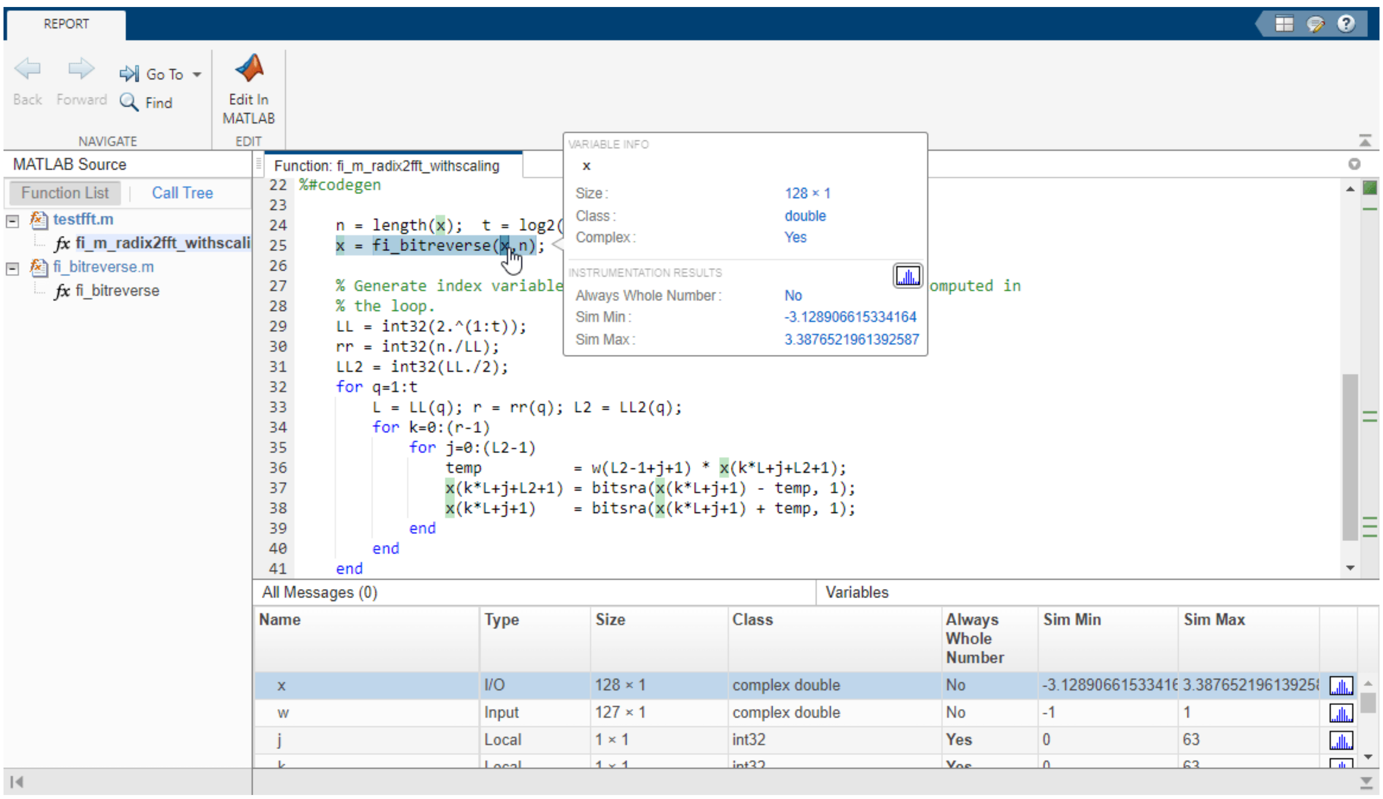
```
buildInstrumentedMex testfft -coder -o testfft_instrumented -args {x,w}
```

Like the fiaccel function, the buildInstrumentedMex function generates a MEX function. To generate C code, use the MATLAB® Coder™ codegen function.

Run a test file to record instrumentation results. Use the showInstrumentedMex function to open the report. To view the simulation minimum and maximum values and whole number status, pause over a variable in the report. You can also see proposed data types for double precision numbers in the table.

```
for i=1:20
   y = testfft_instrumented(randn(size(x)),w);
end
```

```
showInstrumentationResults testfft_instrumented
```



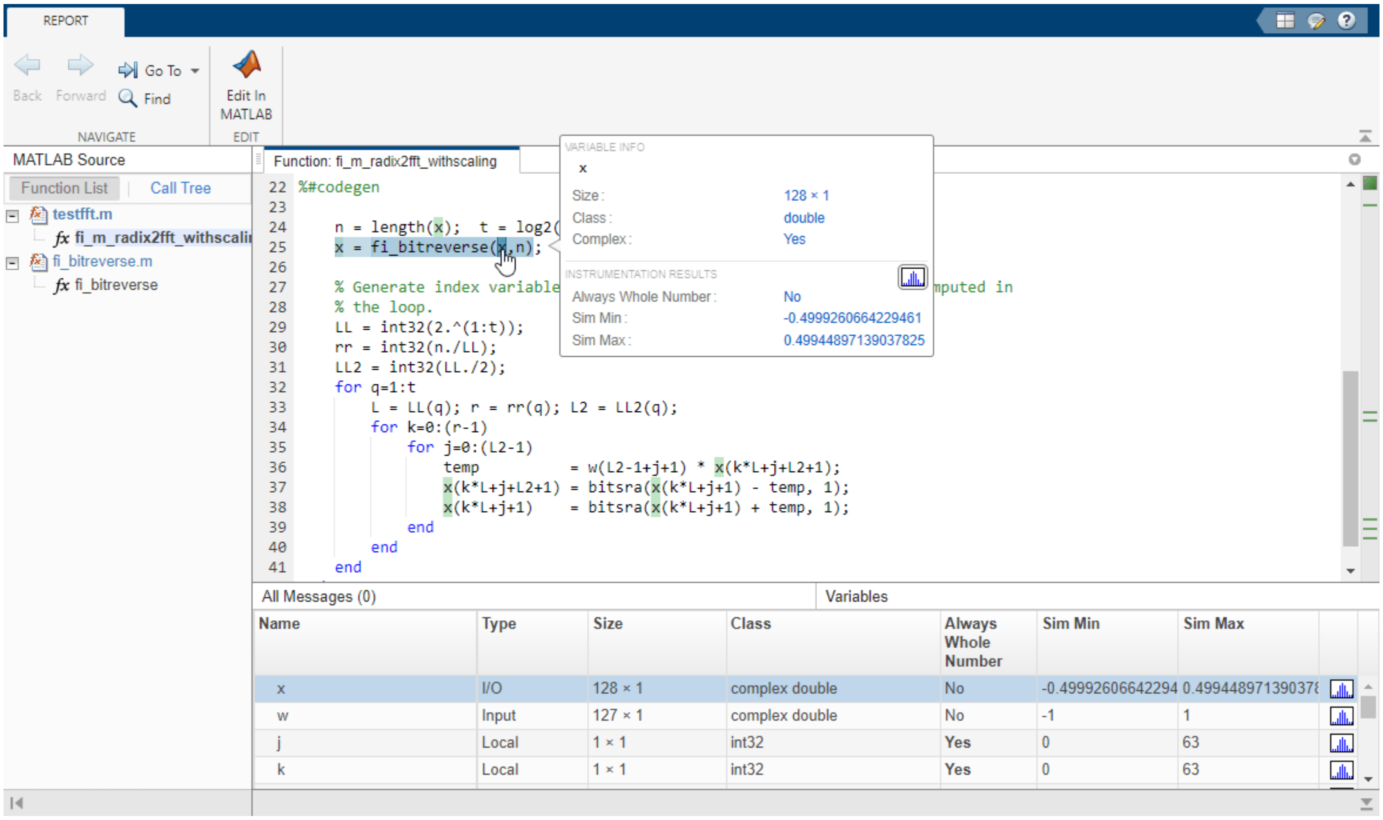
Close the histogram display, then use the clearInstrumentationResults function to clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

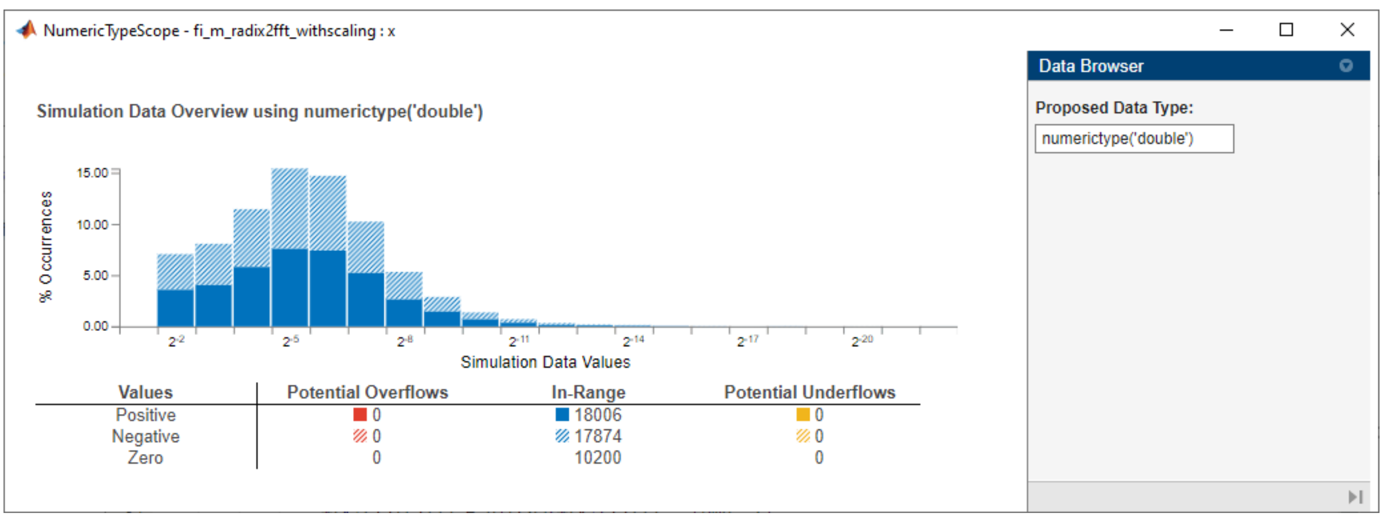
Run a different test bench, then view the new instrumentation results.

```
for i=1:20
   y = testfft_instrumented(cast(rand(size(x))-0.5,'like',x),w);
end
```

```
showInstrumentationResults testfft_instrumented
```



To view the histogram for a variable, click the histogram icon in the **Variables** tab.



Close the histogram display, then use the `clearInstrumentationResults` function to clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

Clear the MEX function.

```
clear testfft_instrumented
```

Create Instrumented MEX Function for Multiple Entry Point Functions

This example shows how to create an instrumented MEX function for multiple entry point functions. Generating a MEX function for multiple entry point functions using the `buildInstrumentedMex` function requires using the `'-coder'` option and a MATLAB® Coder™ license.

Section 1 Title

This example uses the two entry-point functions, `ep1` and `ep2`.

```
function y1 = ep1(u) %#codegen
y1 = u;
end
```

```
function y2 = ep2(u,v) %#codegen
y2 = u + v;
end
```

Generate an instrumented MEX function for the two entry-point functions. Use the `-o` option to specify the name of the MEX function. Use the `-histogram` option to compute histograms. Use the `-coder` option to enable generating multiple entry points with the `buildInstrumentedMex` function.

```
u = 1:100;
v = 5:104;
buildInstrumentedMex -o sharedmex ...
ep1 -args {u} ... % Entry point 1
ep2 -args {u,v} ... % Entry point 2
-histogram -coder
```

```
Code generation successful.
```

Use the generated MEX function to call the first entry-point function.

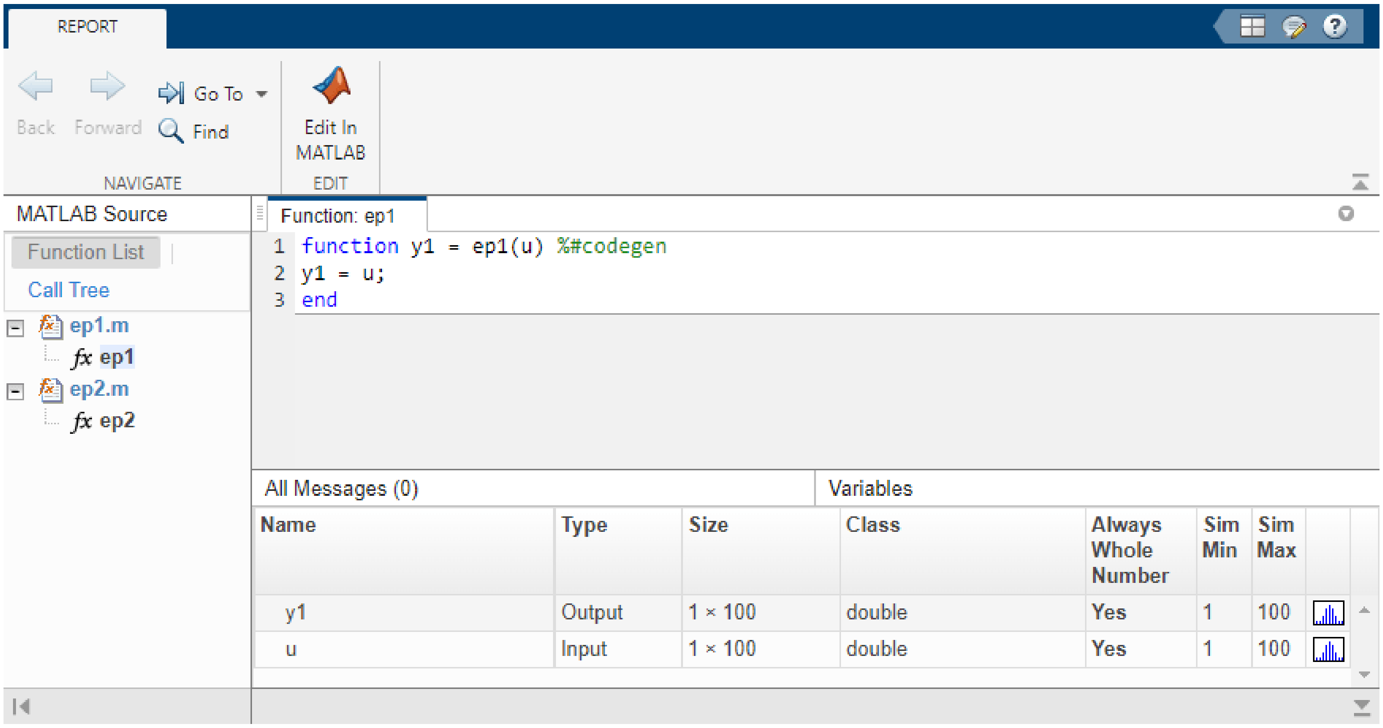
```
y1 = sharedmex('ep1',u);
```

Use the generated MEX function to call the second entry-point function.

```
y2 = sharedmex('ep2',u,v);
```

Show the instrumentation results.

```
showInstrumentationResults sharedmex
```



Input Arguments

`fcn` — Entry-point functions to instrument

function name

MATLAB entry-point functions to be instrumented, specified as a function existing in the current working folder or on the path. The entry-point functions must be suitable for code generation. For more information, see “Make the MATLAB Code Suitable for Code Generation” (MATLAB Coder).

`options` — Compiler options

option value | space delimited list of option values

Choice of compiler options. `buildInstrumentedMex` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the rightmost option prevails.

`-args` *`example_inputs`*	Define the size, class, and complexity of all MATLAB function inputs. Use the values in *`example_inputs`* to define these properties. *`example_inputs`* must be a cell array that specifies the same number and order of inputs as the MATLAB function.

`-coder`	Use MATLAB Coder software to compile the MEX file, instead of the default Fixed-Point Designer `fiaccel` function. This option removes `fiaccel` restrictions and allows for full code generation support. You must have a MATLAB Coder license to use this option.
`-config` *`config_object`*	Specify MEX generation parameters, based on *`config_object`*, defined as a MATLAB variable using `coder.mexconfig`. For example: `cfg = coder.mexconfig;`
`-d` *`out_folder`*	Store generated files in the absolute or relative path specified by *`out_folder`*. If the folder specified by *`out_folder`* does not exist, `buildInstrumentedMex` creates it for you. If you do not specify the folder location, `buildInstrumentedMex` generates files in the default folder: *`fiaccel/mex/fcn`*. *`fcn`* is the name of the MATLAB function specified at the command line. The function does not support the following characters in folder names: asterisk (`*`), question-mark (`?`), dollar (`$`), and pound (`#`).
`-g`	Compiles the MEX function in debug mode, with optimization turned off. If not specified, `buildinstrumentedMex` generates the MEX function in optimized mode.

`-global` *`global_values`*	Specify initial values for global variables in MATLAB file. Use the values in cell array `global_values` to initialize global variables in the function you compile. The cell array should provide the name and initial value of each global variable. You must initialize global variables before compiling with `buildInstrumentedMex`. If you do not provide initial values for global variables using the `-global` option, `buildInstrumentedMex` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `buildInstrumentedMex` generates an error. The generated MEX code and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ.
`-histogram`	Compute the log2 histogram for all named, intermediate and expression values. A histogram column appears in the code generation report table.
`-I` *`include_path`*	Add *`include_path`* to the beginning of the code generation path. `buildInstrumentedMex` searches the code generation path *first* when converting MATLAB code to MEX code.
`-launchreport`	Generate and open a code generation report. If you do not specify this option, `buildInstrumentedMex` generates a report only if error or warning messages occur or you specify the `-report` option.
`-o` *`output_file_name`*	Generate the MEX function with the base name *`output_file_name`* plus a platform-specific extension. *`output_file_name`* can be a file name or include an existing path. If you do not specify an output file name, the base name is *`fcn_mex`*, which allows you to run the original MATLAB function and the MEX function and compare the results.

`-O` *`optimization_option`*	Optimize generated MEX code, based on the value of *`optimization_option`*: • `enable:inline` — Enable function inlining • `disable:inline` — Disable function inlining If not specified, `buildInstrumentedMex` uses inlining for optimization.
`-report`	Generate a code generation report. If you do not specify this option, `buildInstrumentedMex` generates a report only if error or warning messages occur or you specify the `-launchreport` option.

Tips

- You cannot instrument MATLAB functions provided with the software. If your top-level function is such a MATLAB function, nothing is logged. You also cannot instrument scripts.
- Instrumentation results are accumulated every time the instrumented MEX function is called. Use `clearInstrumentationResults` to clear previous results in the log.
- Some coding patterns pass a significant amount of data, but only use a small portion of that data. In such cases, you may see degraded performance when using `buildInstrumentedMex`. In the following pattern, `subfun` only uses one element of input array, `A`. For normal execution, the amount of time to execute `subfun` once remains constant regardless of the size of `A`. The function `topfun` calls `subfun` `N` times, and thus the total time to execute `topfun` is proportional to `N`. When instrumented, however, the time to execute `subfun` once becomes proportional to `N^2`. This change occurs because the minimum and maximum data are calculated over the entire array. When `A` is large, the calculations can lead to significant performance degradation. Therefore, whenever possible, you should pass only the data that the function actually needs.

  ```
  function A = topfun(A)
      N = numel(A);
      for i=1:N
          A(i) = subfun(A,i);
      end
  end
  function b = subfun(A,i)
      b = 0.5 * A(i);
  end

  function A = topfun(A)
      N = numel(A);
      for i=1:N
          A(i) = subfun(A(i));
      end
  end
  function b = subfun(a)
      b = 0.5 * a;
  end
  ```

Version History

Introduced in R2011b

R2023a: Change to default `buildInstrumentedMex` behavior for constant inputs

When the `buildInstrumentedMex` function generates a MEX file, it no longer automatically removes constants from the call to the MEX file. With this change, `buildInstrumentedMex` and `codegen` MEX generation now have the same default behavior with respect to constant inputs.

```
buildInstrumentedMex myfun -args {x,coder.Constant(c)}
myfun_mex(x,c)
```

To revert to the prior behavior, set the `ConstantInputs` property of the `coder.MexConfig` object to `'Remove'`.

```
cfg = coder.MexConfig;
cfg.ConstantInputs = "Remove";
buildInstrumentedMex myfun -args {x,coder.Constant(c)} -config cfg
myfun_mex(x)
```

See Also

`fiaccel` | `clearInstrumentationResults` | `showInstrumentationResults` | `NumericTypeScope` | `codegen` | `mex`

cast

Cast variable to different data type

Syntax

```
b = cast(a,'like',p)
```

Description

`b = cast(a,'like',p)` converts `a` to the same `numerictype`, complexity (real or complex), and `fimath` as `p`. If `a` and `p` are both real, then `b` is also real. Otherwise, `b` is complex.

Examples

Convert an int8 Value to Fixed Point

Define a scalar 8–bit integer.

```
a = int8(5);
```

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Convert `a` to fixed point with `numerictype`, complexity (real or complex), and `fimath` of the specified `fi` object, `p`.

```
b = cast(a, 'like', p)
```

```
b = 
     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Convert an Array to Fixed Point

Define a 2-by-3 matrix of ones.

```
A = ones(2,3);
```

Create a signed `fi` object with word length of `16` and fraction length of `8`.

```
p = fi([],1,16,8);
```

Convert `A` to the same data type and complexity (real or complex) as `p`.

```
B = cast(A,'like',p)
```

```
B = 
     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8
```

Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    for i = 1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818   0.195262145875635   0.097631072937818];
    a = [1.000000000000000  -0.942809041582063   0.333333333333333];

    % Define floating-point types
    T_float.coeffs = double([]);
    T_float.data   = double([]);

    % Create a step input using ones with the
    % floating-point data type
    t = 0:20;
    x_float = ones(size(t),'like',T_float.data);
```

```
    % Initialize the states using zeros with the
    % floating-point data type
    z_float = zeros(1,2,'like',T_float.data);

    % Run the floating-point algorithm
    y_float = my_filter(b,a,x_float,z_float,T_float);

    % Define fixed-point types
    T_fixed.coeffs = fi([],true,8,6);
    T_fixed.data   = fi([],true,8,6);

    % Create a step input using ones with the
    % fixed-point data type
    x_fixed = ones(size(t),'like',T_fixed.data);

    % Initialize the states using zeros with the
    % fixed-point data type
    z_fixed = zeros(1,2,'like',T_fixed.data);

    % Run the fixed-point algorithm
    y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

    % Compare the results
    coder.extrinsic('clf','subplot','plot','legend')
    clf
    subplot(211)
    plot(t,y_float,'co-',t,y_fixed,'kx-')
    legend('Floating-point output','Fixed-point output')
    title('Step response')
    subplot(212)
    plot(t,y_float - double(y_fixed),'rs-')
    legend('Error')
    figure(gcf)
end
```

Input Arguments

a — Variable that you want to cast to a different data type

`fi` object | numeric variable

Variable, specified as a `fi` object or numeric variable.

Complex Number Support: Yes

p — Prototype

`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.

Complex Number Support: Yes

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Version History

Introduced in R2013a

See Also

`ones` | `zeros` | `cast`

Topics

"Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types Using cast and zeros"
"Manual Fixed-Point Conversion Workflow"
"Manual Fixed-Point Conversion Best Practices"

cast64BitFiToInt

Cast `fi` object types that can be exactly represented to a 64-bit integer data type

Syntax

```
y = cast64BitFiToInt(u)
```

Description

`y = cast64BitFiToInt(u)` casts the input `u` to an equivalent 64-bit integer data type when possible.

If the input `u` is a `fi` object that can be represented exactly by an `int64` or `uint64` data type, then the output is this built-in data type. If `u` is a `fi` object that cannot be exactly represented by a built-in data type, or if it is already a built-in data type, then the output is the same as the input.

Examples

Cast a `fi` Object to an Equivalent Integer Type

Use the `castFiToInt` and `cast64BitFiToInt` functions to cast `fi` objects to equivalent integer data types.

Create a signed `fi` variable with a 16-bit word length and zero fraction length. This is equivalent to an `int16` data type. Cast the variable to the equivalent integer data type using the `castFiToInt` function.

```
u = fi(25,1,16,0);
y1 = castFiToInt(u)
```

```
y1 =

  int16

   25
```

The `cast64BitFiToInt` function casts only 64-bit word length `fi` objects with zero fraction length to an equivalent integer data type. All other input data types retain their original data type.

In this example, because the input is not a 64-bit word length `fi`, the output is the same as the input.

```
y2 = cast64BitFiToInt(u)
```

```
y2 =

    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

When you pass a `fi` object with a 64-bit word length and zero fraction length into the `cast64BitFiToInt` function, the output is an `int64`.

```
u = fi(25,1,64,0)
y3 = cast64BitFiToInt(u)
```

```
y3 =

  int64

   25
```

When the input is a `fi` object with a non-zero fraction length, both functions return the original `fi` object because the input cannot be represented by an integer data type.

```
u = fi(pi,1,64,32);
y4 = cast64BitFiToInt(u)
```

```
y4 =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 32
```

```
y5 = castFiToInt(u)
```

```
y5 =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 32
```

Input Arguments

u — Numeric input

scalar | vector | matrix | multidimensional array

Numeric input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `half` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Output Arguments

y — Numeric output

scalar | vector | matrix | multidimensional array

Numeric output, returned as a scalar, vector, matrix, or multidimensional array with the same value and dimensions as the input.

If the input `u` is a `fi` object that can be represented exactly by an `int64` or `uint64` data type, then the output is this built-in data type. If `u` is a `fi` object that cannot be exactly represented by a built-in data type, or if it is already a built-in data type, then the output is the same as the input.

Version History

Introduced in R2020a

See Also

`cast64BitIntToFi` | `castFiToInt` | `castFiToMATLAB` | `castIntToFi`

cast64BitIntToFi

Cast 64-bit integer types to an equivalent `fi` object type

Syntax

```
y = cast64BitIntToFi(u)
```

Description

`y = cast64BitIntToFi(u)` casts the input variable `u` to an equivalent 64-bit `fi` object when the data type of `u` is a 64-bit integer type. Otherwise, the output has the same data type as the input.

Examples

Cast an Integer to a `fi` Object

Use the `castIntToFi` and `cast64BitIntToFi` functions to cast integer data types in your code to equivalent `fi` objects.

Create a variable with a signed 16-bit integer data type. Cast the variable to an equivalent `fi` object using the `castIntToFi` function.

```
u = int16(25);
y1 = castIntToFi(u)
```

```
y1 =

    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

The output `fi` object has the same word length and signedness as the input, and zero fraction length.

The `cast64BitIntToFi` function casts only 64-bit integer data types to an equivalent `fi` object. All other input data types retain their data type.

In this example, because the input is not an `int64` or `uint64` data type, the output remains an `int16`.

```
y2 = cast64BitIntToFi(u)
```

```
y2 =

  int16

   25
```

When you pass an `int64` into the `cast64BitIntToFi` function, the output is a `fi` object with a 64-bit word length and zero fraction length.

```
u = int64(25);
y3 = castIntToFi(u)
```

```
y3 =

    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 0
```

Input Arguments

u — Numeric input
scalar | vector | matrix | multidimensional array

Numeric input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `half` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Output Arguments

y — Numeric output
scalar | vector | matrix | multidimensional array

Numeric output, returned as a scalar, vector, matrix, or multidimensional array with the same value and dimensions as the input.

When the data type of `u` is a 64-bit integer type, the output is a `fi` object with a 64-bit word length, fraction length of zero, and the same signedness as the input. Otherwise, the output has the same data type as the input.

Version History

Introduced in R2020a

See Also

`cast64BitFiToInt` | `castFiToInt` | `castFiToMATLAB` | `castIntToFi`

castFiToInt

Cast fi object to equivalent integer data type

Syntax

`y = castFiToInt(u)`

Description

`y = castFiToInt(u)` casts the input `u` to an equivalent MATLAB integer data type when possible.

If the input `u` is a `fi` object type that can be represented exactly by an integer data type, then the output is this integer data type. If `u` is a `fi` object that cannot be exactly represented by a built-in data type, or if it is already a built-in data type, then the output is the same as the input.

Examples

Cast a `fi` Object to an Equivalent Integer Type

Use the `castFiToInt` and `cast64BitFiToInt` functions to cast `fi` objects to equivalent integer data types.

Create a signed `fi` variable with a 16-bit word length and zero fraction length. This is equivalent to an `int16` data type. Cast the variable to the equivalent integer data type using the `castFiToInt` function.

```
u = fi(25,1,16,0);
y1 = castFiToInt(u)
```

```
y1 =

  int16

   25
```

The `cast64BitFiToInt` function casts only 64-bit word length `fi` objects with zero fraction length to an equivalent integer data type. All other input data types retain their original data type.

In this example, because the input is not a 64-bit word length `fi`, the output is the same as the input.

```
y2 = cast64BitFiToInt(u)
```

```
y2 =

    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

When you pass a `fi` object with a 64-bit word length and zero fraction length into the `cast64BitFiToInt` function, the output is an `int64`.

```
u = fi(25,1,64,0)
y3 = cast64BitFiToInt(u)
```

```
y3 =

  int64

   25
```

When the input is a `fi` object with a non-zero fraction length, both functions return the original `fi` object because the input cannot be represented by an integer data type.

```
u = fi(pi,1,64,32);
y4 = cast64BitFiToInt(u)
```

```
y4 =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 32
```

```
y5 = castFiToInt(u)
```

```
y5 =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 32
```

Input Arguments

u — Numeric input
scalar | vector | matrix | multidimensional array

Numeric input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `half` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Output Arguments

y — Numeric output
scalar | vector | matrix | multidimensional array

Numeric output, returned as a scalar, vector, matrix, or multidimensional array with the same value and dimensions as the input.

Version History

Introduced in R2020a

See Also

`cast64BitFiToInt` | `cast64BitIntToFi` | `castFiToMATLAB` | `castIntToFi`

castFiToMATLAB

Cast `fi` object type to an equivalent built-in MATLAB data type

Syntax

```
y = castFiToMATLAB(u)
```

Description

`y = castFiToMATLAB(u)` casts the input `u` to an equivalent MATLAB built-in data type when possible.

If the input `u` is a `fi` object type that can be represented exactly by a built-in MATLAB data type, then the output is this built-in data type. If `u` is a `fi` object type that cannot be exactly represented by a built-in data type, or if it is already a built-in data type, then the output is the same as the input.

Examples

Cast a `fi` Object to an Equivalent Built-In MATLAB Type

Use the `castFiToMATLAB` function to cast `fi` objects to equivalent built-in MATLAB data types.

Create a signed `fi` variable with a 16-bit word length and zero fraction length. This is equivalent to an `int16` data type. Cast the variable to the equivalent MATLAB data type using the `castFiToMATLAB` function.

```
u = fi(25,1,16,0);
y1 = castFiToMATLAB(u)
```

```
y1 =

  int16

   25
```

When the input is a `fi` object with a non-zero fraction length, the function returns the original `fi` object because the input cannot be represented by a built-in data type.

```
u = fi(pi,1,64,32);
y2 = castFiToMATLAB(u)
```

```
y2 =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 32
```

When the input is a double-precision `fi` object, the function returns a double with the same value.

```
T = numerictype('Double');
u = fi(25,T)

u = 

    25

          DataTypeMode: Double

y3 = castFiToMATLAB(u)
class(y3)

y3 =

    25

ans =

    'double'
```

Input Arguments

u — Numeric input
scalar | vector | matrix | multidimensional array

Numeric input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `half` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Output Arguments

y — Numeric output
scalar | vector | matrix | multidimensional array

Numeric output, returned as a scalar, vector, matrix, or multidimensional array with the same value and dimensions as the input.

If the input `u` is a `fi` object that can be represented exactly by a built-in MATLAB data type, then the output is this built-in data type. If `u` is a `fi` object that cannot be exactly represented by a built-in data type, or if it is already a built-in data type, then the output is the same as the input.

Version History

Introduced in R2020a

See Also

`cast64BitFiToInt` | `cast64BitIntToFi` | `castFiToInt` | `castIntToFi`

castIntToFi

Cast an integer data type to equivalent fi type

Syntax

```
y = castIntToFi(u)
```

Description

y = castIntToFi(u) casts the input variable u to an equivalent fi object when u is one of the built-in MATLAB integer data types (int8, uint8, int16, uint16, int32, uint32, int64, uint64).

When u is not one of the built-in integer data types, the output has the same data type as the input.

Examples

Cast an Integer to a fi Object

Use the castIntToFi and cast64BitIntToFi functions to cast integer data types in your code to equivalent fi objects.

Create a variable with a signed 16-bit integer data type. Cast the variable to an equivalent fi object using the castIntToFi function.

```
u = int16(25);
y1 = castIntToFi(u)
```

```
y1 = 

    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

The output fi object has the same word length and signedness as the input, and zero fraction length.

The cast64BitIntToFi function casts only 64-bit integer data types to an equivalent fi object. All other input data types retain their data type.

In this example, because the input is not an int64 or uint64 data type, the output remains an int16.

```
y2 = cast64BitIntToFi(u)
```

```
y2 = 

  int16

   25
```

When you pass an `int64` into the `cast64BitIntToFi` function, the output is a `fi` object with a 64-bit word length and zero fraction length.

```
u = int64(25);
y3 = castIntToFi(u)
```

```
y3 =

    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 64
        FractionLength: 0
```

Input Arguments

u — Numeric input
scalar | vector | matrix | multidimensional array

Numeric input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `half` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Output Arguments

y — Fixed-point output
`fi` object | scalar | vector | matrix | multidimensional array

Numeric output, returned as a scalar, vector, matrix, or multidimensional array with the same value and dimensions as the input.

When the data type of `u` is an integer type, the output is a `fi` object with the same word length and signedness as the input, and a fraction length of zero. Otherwise, the output has the same data type as the input.

Version History

Introduced in R2020a

See Also

`cast64BitFiToInt` | `cast64BitIntToFi` | `castFiToInt` | `castFiToMATLAB`

ceil

Rounds toward positive infinity

Syntax

```
y = ceil(a)
```

Description

`y = ceil(a)` rounds `fi` object `a` to the nearest integer in the direction of positive infinity and returns the result in `fi` object `y`.

Examples

Use `ceil` on a Signed `fi` Object

The following example demonstrates how the `ceil` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi,1,8,3)
```

```
a = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

```
y = ceil(a)
```

```
y = 
     4

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

The following example demonstrates how the `ceil` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a = 
    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
y = ceil(a)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Compare Rounding Methods

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

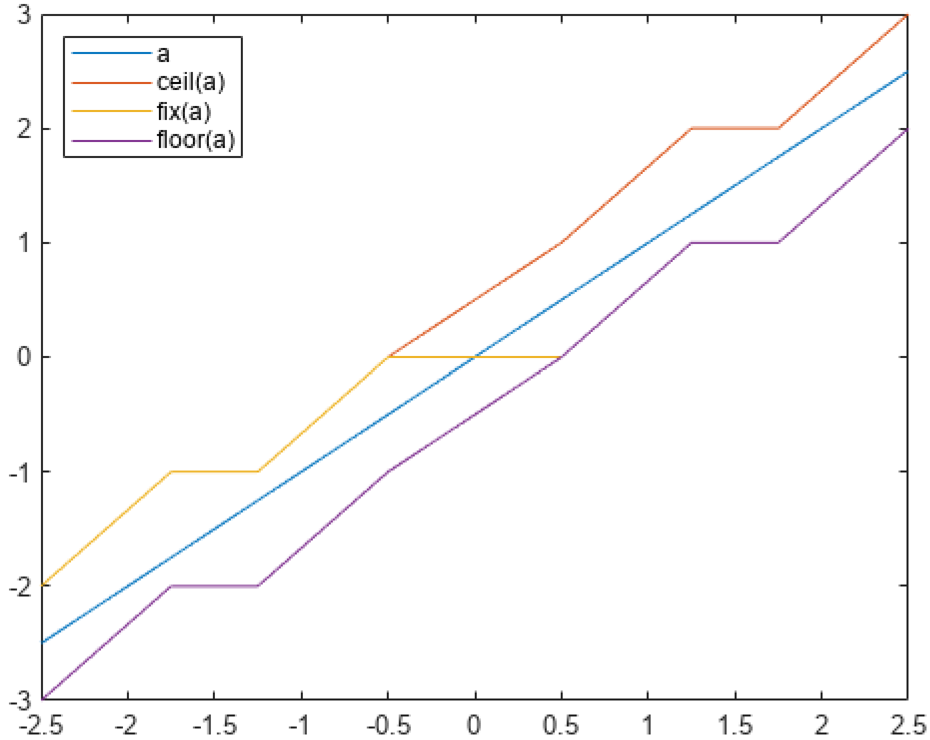
- The `ceil` function rounds values to the nearest integer toward positive infinity.
- The `fix` function rounds values to the nearest integer toward zero.
- The `floor` function rounds values to the nearest integer toward negative infinity.

This example illustrates these differences for a given `fi` input object `a`.

```
a = fi([-2.5,-1.75,-1.25,-0.5,0.5,1.25,1.75,2.5]');
y = [a ceil(a) fix(a) floor(a)]
```

```
y = 
   -2.5000   -2.0000   -2.0000   -3.0000
   -1.7500   -1.0000   -1.0000   -2.0000
   -1.2500   -1.0000   -1.0000   -2.0000
   -0.5000         0         0   -1.0000
    0.5000    1.0000         0         0
    1.2500    2.0000    1.0000    1.0000
    1.7500    2.0000    1.0000    1.0000
    2.5000    3.0000    2.0000    2.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
plot(a,y); legend('a','ceil(a)','fix(a)','floor(a)','location','NW');
```



Input Arguments

a — Input `fi` array
scalar | vector | matrix | multidimensional array

Input `fi` array, specified as scalar, vector, matrix, or multidimensional array.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`ceil` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Data Types: `fi`
Complex Number Support: Yes

Algorithms

- `y` and `a` have the same `fimath` object and `DataType` property.
- When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

- When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

Version History

Introduced in R2008a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`convergent` | `fix` | `floor` | `nearest` | `round`

ceilDiv

Round the result of division toward positive infinity

Syntax

```
y = ceilDiv(x,d)
y = ceilDiv(x,d,m)
```

Description

`y = ceilDiv(x,d)` returns the result of `x/d` rounded to the nearest integer value in the direction of positive infinity.

`y = ceilDiv(x,d,m)` returns the result of `x/d` rounded to the nearest multiple of `m` in the direction of positive infinity.

The datatype of `y` is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of `x`, and the values of `d` and `m`.

Examples

Divide and Round to Ceil

Perform a division operation and round to the nearest integer value in the direction of positive infinity.

```
ceilDiv(int16(201),10)
```

```
ans = 
    21

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Perform a division operation and round to the nearest multiple of 5 in the direction of positive infinity.

```
ceilDiv(int16(201),10,5)
```

```
ans = 
    25

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 14
        FractionLength: 0
```

Divide and Generate Code

Define a function that uses `ceilDiv`.

```
function y = ceilDiv_example(x,d)
y = ceilDiv(x,d);
end
```

Define inputs and execute the function in MATLAB®.

```
x = fi(pi);
d = fi(2);
y = ceilDiv_example(x,d)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

To generate code for this function, the denominator `d` must be defined as a constant.

```
codegen ceilDiv_example -args {x, coder.Constant(d)}
```

```
Code generation successful.
```

Alternatively, you can define the denominator, `d`, as constant in the body of the code.

```
function y = ceilDiv10(x)
y = ceilDiv(x,10);
end
```

```
x = fi(5*pi);
y = ceilDiv10(x)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

```
codegen ceilDiv10 -args {x}
```

```
Code generation successful.
```

Input Arguments

x — Dividend
scalar

Dividend, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

d — Divisor
scalar

Divisor, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

m — Value to round to nearest multiple of
1 (default) | scalar

Value to round to nearest multiple of, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Output Arguments

y — Result of division and round to ceiling
scalar

Result of division and round to ceiling, returned as a scalar.

The datatype of `y` is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of `x`, and the values of `d` and `m`.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

To generate code, the denominator `d` must be declared as constant.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also

`fixDiv` | `floorDiv` | `nearestDiv`

clearInstrumentationResults

Clear results logged by instrumented, compiled C code function

Syntax

```
clearInstrumentationResults('mex_fcn')
clearInstrumentationResults mex_fcn
clearInstrumentationResults all
```

Description

`clearInstrumentationResults('mex_fcn')` clears the results logged from calling the instrumented MEX function, `mex_fcn`.

`clearInstrumentationResults mex_fcn` is an alternative syntax for clearing the log.

`clearInstrumentationResults all` clears the results from all instrumented MEX functions.

Examples

Create Instrumented MEX Function

This example shows how to create an instrumented MEX function, run a test bench, then view logged results.

Define prototype input arguments.

```
n = 128;
x = complex(zeros(n,1));
w = fi_radix2twiddles(n);
```

Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name. Use the `-histogram` option to compute histograms.

If you have a MATLAB® Coder™ license, you can also add the `-coder` option. In this case,

```
buildInstrumentedMex testfft -o testfft_instrumented -args {x,coder.Constant(w)} -histogram
```

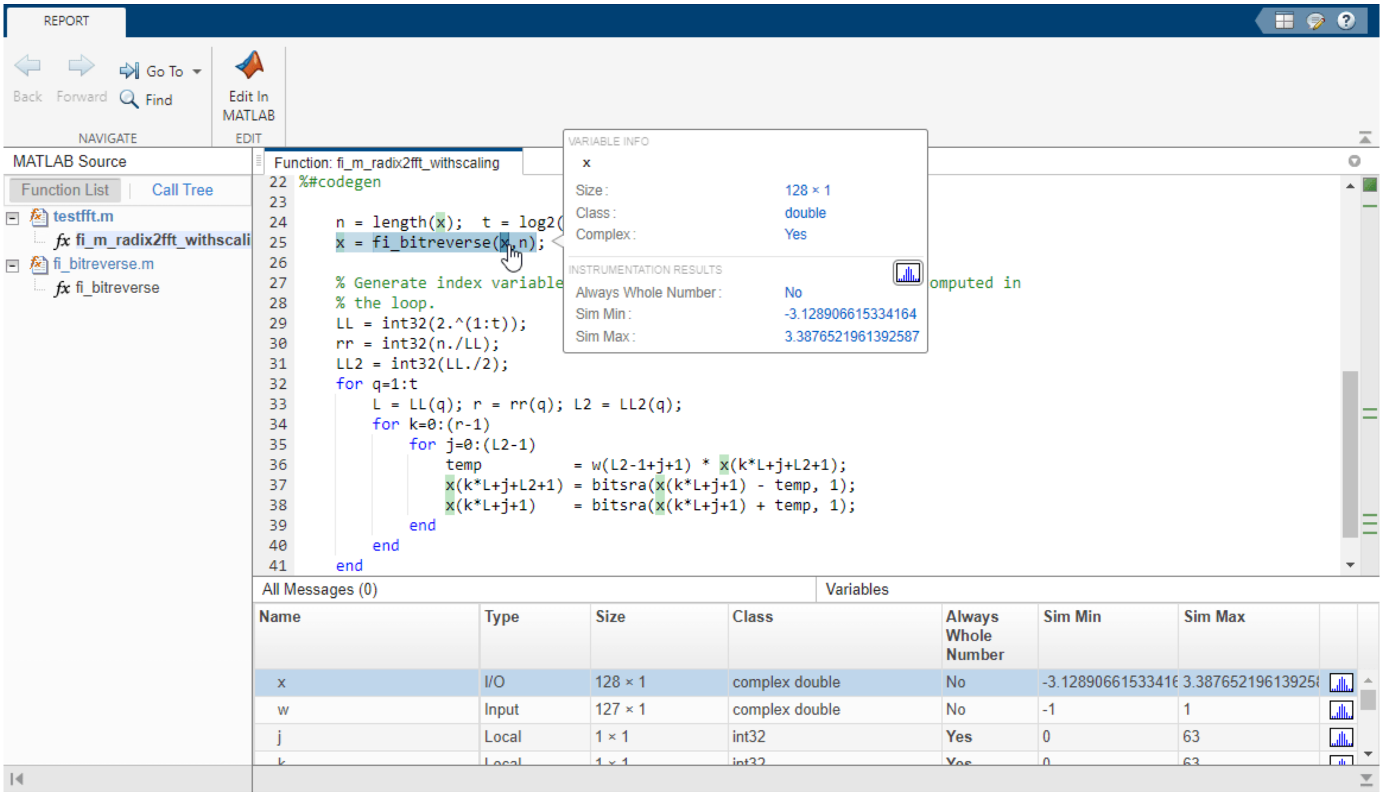
If you have a MATLAB® Coder™ license, you can also add the `-coder` option. For example,

```
buildInstrumentedMex testfft -coder -o testfft_instrumented -args {x,w}
```

Like the `fiaccel` function, the `buildInstrumentedMex` function generates a MEX function. To generate C code, use the MATLAB® Coder™ `codegen` function.

Run a test file to record instrumentation results. Use the `showInstrumentedMex` function to open the report. To view the simulation minimum and maximum values and whole number status, pause over a variable in the report. You can also see proposed data types for double precision numbers in the table.

```
for i=1:20
   y = testfft_instrumented(randn(size(x)),w);
end

showInstrumentationResults testfft_instrumented
```

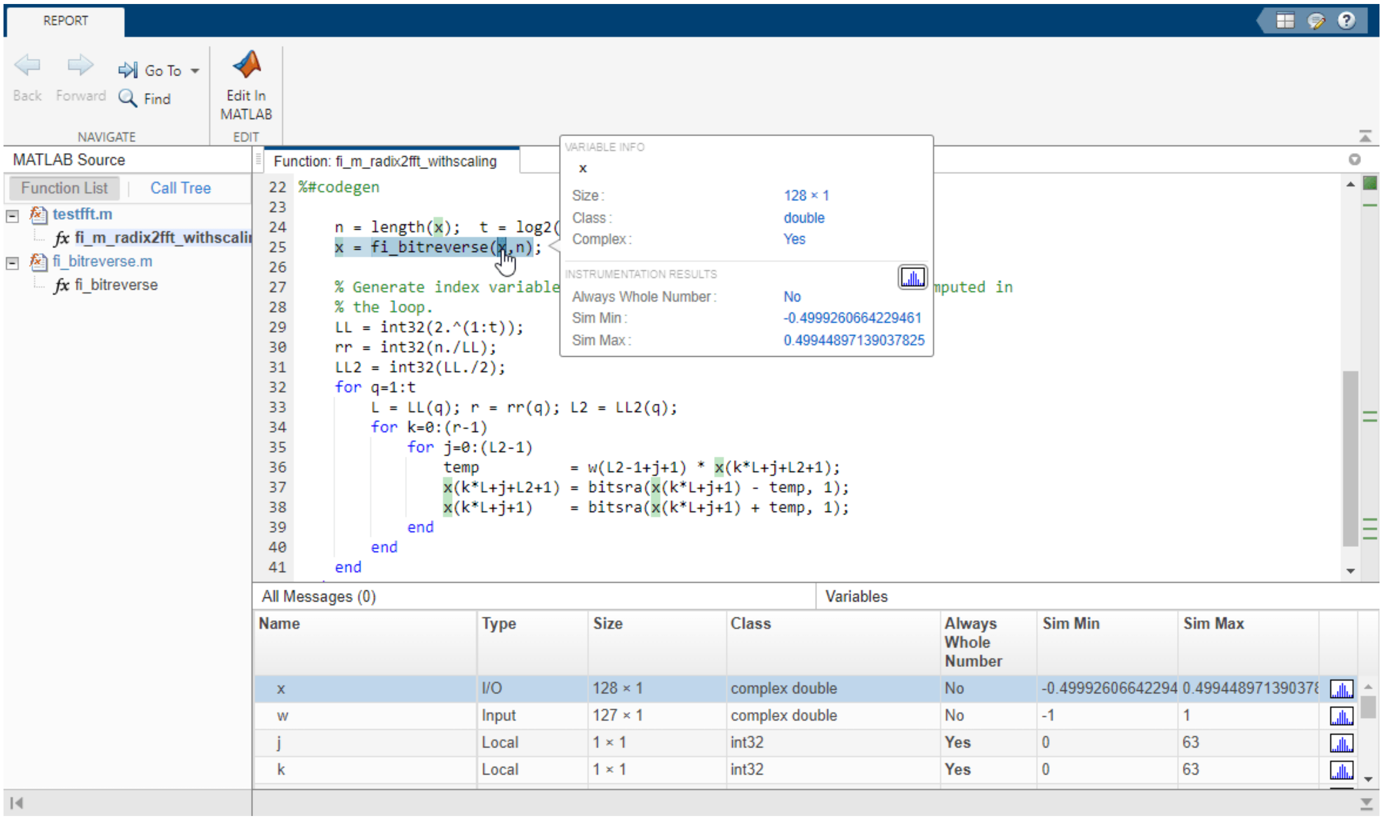


Close the histogram display, then use the `clearInstrumentationResults` function to clear the results log.

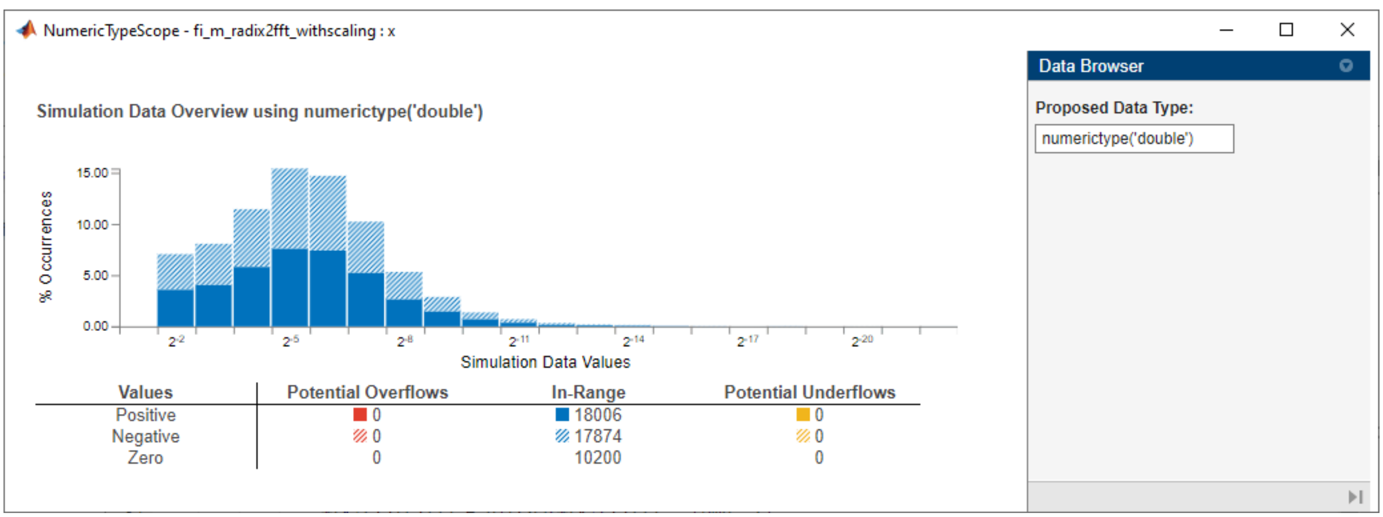
```
clearInstrumentationResults testfft_instrumented
```

Run a different test bench, then view the new instrumentation results.

```
for i=1:20
   y = testfft_instrumented(cast(rand(size(x))-0.5,'like',x),w);
end

showInstrumentationResults testfft_instrumented
```



To view the histogram for a variable, click the histogram icon in the **Variables** tab.



Close the histogram display, then use the `clearInstrumentationResults` function to clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

Clear the MEX function.

```
clear testfft_instrumented
```

Input Arguments

`mex_fcn` — Instrumented MEX function
instrumented MEX function

Instrumented MEX function created using `buildInstrumentedMex`.

Version History

Introduced in R2011b

See Also

`fiaccel` | `showInstrumentationResults` | `buildInstrumentedMex` | `codegen` | `mex`

coder.approximation

Create function replacement configuration object

Syntax

```
q = coder.approximation(function_name)
q = coder.approximation('Function',function_name,Name,Value)
```

Description

`q = coder.approximation(function_name)` creates a function replacement configuration object for use during code generation or fixed-point conversion. The configuration object specifies how to create a lookup table approximation for the MATLAB function specified by `function_name`. To associate this approximation with a `coder.FixptConfig` object for use with the`fiaccel` function, use the `coder.FixptConfig` configuration object `addApproximation` method.

Use this syntax only for the functions that `coder.approximation` can replace automatically. These functions are listed in the `function_name` argument description.

`q = coder.approximation('Function',function_name,Name,Value)` creates a function replacement configuration object using additional options specified by one or more name-value pair arguments.

Examples

Replace `log` Function with Default Lookup Table

Create a function replacement configuration object using the default settings. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('log');
```

Replace `log` Function with Uniform Lookup Table

Create a function replacement configuration object. Specify the input range and prefix to add to the replacement function name. The resulting lookup table in the generated code uses 1000 points.

```
logAppx = coder.approximation('Function','log','InputRange',[0.1,1000],...
'FunctionNamePrefix','log_replace_');
```

Replace `log` Function with Optimized Lookup Table

Create a function replacement configuration object using the `'OptimizeLUTSize'` option to specify to replace the `log` function with an optimized lookup table. The resulting lookup table in the generated code uses less than the default number of points.

```
    logAppx = coder.approximation('Function','log','OptimizeLUTSize', true,...
'InputRange',[0.1,1000],'InterpolationDegree',1,'ErrorThreshold',1e-3,...
'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Replace Custom Function with Optimized Lookup Table

Create a function replacement configuration object that specifies to replace the custom function, `saturateExp`, with an optimized lookup table.

Create a custom function, `saturateExp`.

```
saturateExp = @(x) 1/(1+exp(-x));
```

Create a function replacement configuration object that specifies to replace the `saturateExp` function with an optimized lookup table. Because the `saturateExp` function is not listed as a function for which `coder.approximation` can generate an approximation automatically, you must specify the `CandidateFunction` property.

```
saturateExp = @(x) 1/(1+exp(-x));
custAppx = coder.approximation('Function','saturateExp',...
'CandidateFunction', saturateExp,...
'NumberOfPoints',50,'InputRange',[0,10]);
```

Input Arguments

`function_name` — Name of the function to replace
'acos' | 'acosd' | 'acosh' | 'acoth' | 'asin' | 'asind' | 'asinh' | 'atan' | 'atand' | 'atanh' | 'cos' | 'cosd' | 'cosh' | 'erf ' | 'erfc' | 'exp' | 'log' | 'normcdf' | 'reallog' | 'realsqrt' | 'reciprocal' | 'rsqrt' | 'sin' | 'sinc' | 'sind' | 'sinh' | 'sqrt' | 'tan' | 'tand'

Name of function to replace, specified as a string. The function must be one of the listed functions.

Example: `'sqrt'`

Data Types: `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Function', 'log'`

`Architecture` — Architecture of lookup table approximation
`'LookupTable'` (default) | `'Flat'`

Architecture of the lookup table approximation, specified as the comma-separated pair consisting of `'Architecture'` and a string. Use this argument when you want to specify the architecture for the lookup table. The `Flat` architecture does not use interpolation.

Data Types: `char`

CandidateFunction — Function handle of the replacement function
function handle | string

Function handle of the replacement function, specified as the comma-separated pair consisting of 'CandidateFunction' and a function handle or string referring to a function handle. Use this argument when the function that you want to replace is not listed under function_name. Specify the function handle or string referring to a function handle of the function that you want to replace. You can define the function in a file or as an anonymous function.

If you do not specify a candidate function, then the function you chose to replace using the Function property is set as the CandidateFunction.

Example: 'CandidateFunction', @(x) (1./(1+x))

Data Types: function_handle | char

ErrorThreshold — Error threshold value used to calculate optimal lookup table size
0.001 (default) | nonnegative scalar

Error threshold value used to calculate optimal lookup table size, specified as the comma-separated pair consisting of 'ErrorThreshold' and a nonnegative scalar. If 'OptimizeLUTSize' is true, this argument is required.

Function — Name of function to replace with a lookup table approximation
function_name

Name of function to replace with a lookup table approximation, specified as the comma-separated pair consisting of 'Function' and a string. The function must be continuous and stateless. If you specify one of the functions that is listed under function_name, the conversion process automatically provides a replacement function. Otherwise, you must also specify the 'CandidateFunction' argument for the function that you want to replace.

Example: 'Function','log'

Example: 'Function', 'my_log','CandidateFunction',@my_log

Data Types: char

FunctionNamePrefix — Prefix for generated fixed-point function names
'replacement_' (default) | string

Prefix for generated fixed-point function names, specified as the comma-separated pair consisting of 'FunctionNamePrefix' and a string. The name of a generated function consists of this prefix, followed by the original MATLAB function name.

Example: ‘log_replace_’

InputRange — Range over which to replace the function
[ ] (default) | 2x1 row vector | 2xN matrix

Range over which to replace the function, specified as the comma-separated pair consisting of 'InputRange' and a 2-by-1 row vector or a 2-by-*N* matrix.

Example: [-1 1]

InterpolationDegree — Interpolation degree
1 (default) | 0 | 2 | 3

Interpolation degree, specified as the comma-separated pair consisting of `'InterpolationDegree'` and`1` (linear), `0` (none), `2` (quadratic), or `3` (cubic).

`NumberOfPoints` — Number of points in lookup table
`1000` (default) | positive integer

Number of points in lookup table, specified as the comma-separated pair consisting of `'NumberOfPoints'` and a positive integer.

`OptimizeIterations` — Number of iterations
25 (default) | positive integer

Number of iterations to run when optimizing the size of the lookup table, specified as the comma-separated pair consisting of `'OptimizeIterations'` and a positive integer.

`OptimizeLUTSize` — Optimize lookup table size
`false` (default) | `true`

Optimize lookup table size, specified as the comma-separated pair consisting of `'OptimizeLUTSize'` and a logical value. Setting this property to `true` generates an area-optimal lookup table, that is, the lookup table with the minimum possible number of points. This lookup table is optimized for size, but might not be speed efficient.

`PipelinedArchitecture` — Option to enable pipelining
`false` (default) | `true`

Option to enable pipelining, specified as the comma-separated pair consisting of `'PipelinedArchitecture'` and a logical value.

Output Arguments

q — Function replacement configuration object, returned as a `coder.mathfcngenerator.LookupTable` or a `coder.mathfcngenerator.Flat` configuration object
`coder.mathfcngenerator.LookupTable` configuration object |
`coder.mathfcngenerator.Flat` configuration object

Function replacement configuration object that specifies how to create an approximation for a MATLAB function. Use the `coder.FixptConfig` configuration object `addApproximation` method to associate this configuration object with a `coder.FixptConfig` object. Then use the `fiaccel` function `-float2fixed` option with `coder.FixptConfig` to convert floating-point MATLAB code to fixed-point MATLAB code.

Property	Default Value
`Auto-replace function`	`''`
`InputRange`	`[]`
`FunctionNamePrefix`	`'replacement_'`
`Architecture`	`LookupTable` (read only)
`NumberOfPoints`	`1000`
`InterpolationDegree`	`1`

Property	Default Value
`ErrorThreshold`	`0.001`
`OptimizeLUTSize`	`false`
`OptimizeIterations`	`25`

Version History

Introduced in R2014b

See Also

Classes
`coder.FixPtConfig`

Functions
`fiaccel`

Topics
"Replace the exp Function with a Lookup Table"
"Replace a Custom Function with a Lookup Table"
"Replacing Functions Using Lookup Table Approximations"

coder.allowpcode

Package: coder

Control code generation from P-code files

Syntax

```
coder.allowpcode('plain')
```

Description

`coder.allowpcode('plain')` allows you to generate P-code files that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as P-code files that provide code generation optimizations.

Call this function in the top-level function before control-flow statements, such as `if`, `while`, `switch`, and function calls.

MATLAB functions can call P-code. When the `.m` and `.p` versions of a file exist in the same folder, the P-code file takes precedence.

`coder.allowpcode` is ignored outside of code generation.

Examples

Generate optimized embeddable code from P-code file

Write a function `p_abs` that returns the absolute value of its input:

```
function out = p_abs(in)   %#codegen
% The directive %#codegen indicates that the function
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);
```

Generate P-code file. In the MATLAB Command Window, enter:

```
pcode p_abs
```

The P-code file, `p_abs.p`, appears in the current folder.

Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license).

```
codegen p_abs -args { int32(0) }
```

`codegen` generates a MEX function in the current folder.

If you have MATLAB Coder, generate embeddable C code for `p_abs.p`.

```
codegen p_abs -config:lib -args { int32(0) };
```

`codegen` generates C library code in the `codegen\lib\p_abs` folder.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

pcode | codegen

coder.ArrayType class

Package: coder
Superclasses: coder.Type

Represent set of MATLAB arrays acceptable for input specification

Description

Objects of the coder.ArrayType class specify array types that the generated code accepts. Use objects of this class only with the -args option of the fiaccel function. Do not pass as an input to a generated MEX function.

Creation

Note You can create and edit coder.Type objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

coder.ArrayType is an abstract class. You cannot create instances of this class directly. You can create coder.EnumType, coder.FiType, coder.PrimitiveType, and coder.StructType objects that derive from this class.

Properties

ClassName — Value class name
coder.EnumType | coder.FiType | coder.PrimitiveType | coder.StructType

Value class name, specified as one of these class object names.

- coder.EnumType
- coder.FiType
- coder.PrimitiveType
- coder.StructType

SizeVector — Upper bound on array size
positive integer

Upper bound on array size, specified as a positive integer.

VariableDims — Option to specify variable-size
1 | 0

Option to specify whether each dimension of the array has a variable size specified as a boolean vector. A value of 1 indicates that the dimension has variable-size. A value of 0 indicates that the dimension has fixed-size

Version History

Introduced in R2011a

See Also

`coder.ClassType` | `coder.Type` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.CellType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `fiaccel`

Topics

"Create and Edit Input Types by Using the Coder Type Editor"

coder.config

Create configuration object for fixed-point or single-precision conversion

Syntax

```
config_obj = coder.config('fixpt')
config_obj = coder.config('single')
```

Description

`config_obj = coder.config('fixpt')` creates a `coder.FixptConfig` configuration object. Use this object with the `fiaccel` function when converting floating-point MATLAB code to fixed-point MATLAB code.

`config_obj = coder.config('single')` creates a `coder.SingleConfig` configuration object for use with the `convertToSingle` function when generating single-precision MATLAB code from double-precision MATLAB code.

Examples

Convert Floating-Point MATLAB Code to Fixed Point

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccel -float2fixed fixptcfg dti
```

Convert Double-Precision MATLAB Code to Single-Precision MATLAB Code

Create a `coder.SingleConfig` object, `scfg`.

```
scfg = coder.config('single');
```

Set the test bench name. In this example, the test bench function name is `myfun_test`. Enable numerics testing and data logging for comparison plotting of input and output variables.

```
scfg.TestBenchName = 'myfun_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

Convert the double-precision MATLAB code to single-precision MATLAB code. In this example, the MATLAB function name is `myfun`.

```
convertToSingle -config scfg myfun
```

Version History

Introduced in R2014b

See Also

`coder.FixPtConfig` | `fiaccel` | `coder.SingleConfig` | `convertToSingle`

coder.const

Fold expressions into constants in generated code

Syntax

```
out = coder.const(expression)
[out1,...,outN] = coder.const(handle,arg1,...,argN)
```

Description

`out = coder.const(expression)` evaluates `expression` and replaces `out` with the result of the evaluation in generated code.

`[out1,...,outN] = coder.const(handle,arg1,...,argN)` evaluates the multi-output function having handle `handle`. It then replaces `out1,...,outN` with the results of the evaluation in the generated code.

Examples

Specify Constants in Generated Code

This example shows how to specify constants in generated code using `coder.const`.

Write a function `AddShift` that takes an input `Shift` and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. `AddShift` generates this vector.

```
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
```

Generate code for `AddShift` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator produces code for creating the vector. It adds `Shift` to each element of the vector during vector creation. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int k;
  for (k = 0; k < 10; k++) {
    y[k] = (double)((1 + k) * (1 + k)) + Shift;
  }
}
```

Replace the expression `(1:10).^2` with `coder.const((1:10).^2)`, and then generate code for `AddShift` again using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds `Shift` to each element of this vector. The definition of `AddShift` in generated code looks as follows:

```
void AddShift(double Shift, double y[10])
{
  int i;
  static const signed char iv[10] = { 1, 4, 9, 16, 25, 36,
                                       49, 64, 81, 100 };

  for (i = 0; i < 10; i++) {
    y[i] = (double)iv[i] + Shift;
  }
}
```

Create Lookup Table in Generated Code

This example shows how to fold a user-written function into a constant in generated code.

Write a function `getsine` that takes an input `index` and returns the element referred to by `index` from a lookup table of sines. The function `getsine` creates the lookup table using another function `gettable`.

```
function y = getsine(index) %#codegen
  assert(isa(index, 'int32'));
  persistent tbl;
  if isempty(tbl)
           tbl = gettable(1024);
  end
  y = tbl(index);

function y = gettable(n)
      y = zeros(1,n);
      for i = 1:n
         y(i) = sin((i-1)/(2*pi*n));
      end
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

```
codegen -config:lib -launchreport getsine -args int32(0)
```

The generated code contains instructions for creating the lookup table.

Replace the statement:

```
tbl = gettable(1024);
```

with:

```
tbl = coder.const(gettable(1024));
```

Generate code for `getsine` using an argument of type `int32`. Open the Code Generation Report.

The generated code contains the lookup table itself. `coder.const` forces the expression `gettable(1024)` to be evaluated during code generation. The generated code does not contain instructions for the evaluation. The generated code contains the result of the evaluation itself.

Specify Constants in Generated Code Using Multi-Output Function

This example shows how to specify constants in generated code using a multi-output function in a `coder.const` statement.

Write a function `MultiplyConst` that takes an input `factor` and multiplies every element of two vectors `vec1` and `vec2` with `factor`. The function generates `vec1` and `vec2` using another function `EvalConsts`.

```
function [y1,y2] = MultiplyConst(factor) %#codegen
  [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
  y1=vec1.*factor;
  y2=vec2.*factor;

function [f1,f2]=EvalConsts(z,n)
  f1=z.^(2*n)/factorial(2*n);
  f2=z.^(2*n+1)/factorial(2*n+1);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator produces code for creating the vectors.

Replace the statement

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

with

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:lib -launchreport MultiplyConst -args 0
```

The code generator does not generate code for creating the vectors. Instead, it calculates the vectors and specifies the calculated vectors in generated code.

Read Constants by Processing XML File

This example shows how to call an extrinsic function using `coder.const`.

Write an XML file `MyParams.xml` containing the following statements:

```
<params>
    <param name="hello" value="17"/>
    <param name="world" value="42"/>
</params>
```

Save `MyParams.xml` in the current folder.

Write a MATLAB function `xml2struct` that reads an XML file. The function identifies the XML tag `param` inside another tag `params`.

After identifying `param`, the function assigns the value of its attribute `name` to the field name of a structure `s`. The function also assigns the value of attribute `value` to the value of the field.

```
function s = xml2struct(file)

s = struct();
doc = xmlread(file);
els = doc.getElementsByTagName('params');
for i = 0:els.getLength-1
    it = els.item(i);
    ps = it.getElementsByTagName('param');
    for j = 0:ps.getLength-1
        param = ps.item(j);
        paramName = char(param.getAttribute('name'));
        paramValue = char(param.getAttribute('value'));
        paramValue = evalin('base', paramValue);
        s.(paramName) = paramValue;
    end
end
```

Save `xml2struct` in the current folder.

Write a MATLAB function `MyFunc` that reads the XML file `MyParams.xml` into a structure `s` using the function `xml2struct`. Declare `xml2struct` as extrinsic using `coder.extrinsic` and call it in a `coder.const` statement.

```
function y = MyFunc(u) %#codegen
  assert(isa(u, 'double'));
  coder.extrinsic('xml2struct');
  s = coder.const(xml2struct('MyParams.xml'));
  y = s.hello + s.world + u;
```

Generate code for `MyFunc` using the `codegen` command. Open the Code Generation Report.

```
codegen -config:dll -launchreport MyFunc -args 0
```

The code generator executes the call to `xml2struct` during code generation. It replaces the structure fields `s.hello` and `s.world` with the values 17 and 42 in generated code.

Input Arguments

expression — MATLAB expression or user-written function

expression with constants | single-output function with constant arguments

MATLAB expression or user-defined single-output function.

The expression must have compile-time constants only. The function must take constant arguments only. For instance, the following code leads to a code generation error, because `x` is not a compile-time constant.

```
function y=func(x)
   y=coder.const(log10(x));
```

To fix the error, assign x to a constant in the MATLAB code. Alternatively, during code generation, you can use coder.Constant to define input type as follows:

```
codegen -config:lib func -args coder.Constant(10)
```

Example: 2*pi, factorial(10)

handle — Function handle
function handle

Handle to built-in or user-written function.

Example: @log, @sin

Data Types: function_handle

arg1,...,argN — Arguments to the function with handle handle
function arguments that are constants

Arguments to the function with handle handle.

The arguments must be compile-time constants. For instance, the following code leads to a code generation error, because x and y are not compile-time constants.

```
function y=func(x,y)
   y=coder.const(@nchoosek,x,y);
```

To fix the error, assign x and y to constants in the MATLAB code. Alternatively, during code generation, you can use coder.Constant to define input type as follows:

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

Output Arguments

out — Value of expression
value of the evaluated expression

Value of expression. In the generated code, MATLAB Coder replaces occurrences of out with the value of expression.

out1,...,outN — Outputs of the function with handle handle
values of the outputs of the function with handle handle

Outputs of the function with handle handle. MATLAB Coder evaluates the function and replaces occurrences of out1,...,outN with constants in the generated code.

Tips

- When possible, the code generator constant-folds expressions automatically. Typically, automatic constant-folding occurs for expressions with scalars only. Use coder.const when the code generator does not constant-fold expressions on its own.
- When constant-folding computationally intensive function calls, to reduce code generation time, make the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

```
function j = fcn(z)
zTable = coder.const(0:0.01:100);
jTable = coder.const(feval('besselj',3,zTable));
j = interp1(zTable,jTable,z);
end
```

See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

- If `coder.const` is unable to constant-fold a function call, try to force constant-folding by making the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

  ```
  function yi = fcn(xi)
  y = coder.const(feval('rand',1,100));
  yi = interp1(y,xi);
  end
  ```

  See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

Version History

Introduced in R2013b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Topics

“Fold Function Calls into Constants” (MATLAB Coder)
“Use coder.const with Extrinsic Function Calls” (MATLAB Coder)

coder.Constant class

Package: `coder`
Superclasses: `coder.Type`

Specification of constant value for code generation

Description

Use a `coder.Constant` object to define input values that are constant during code generation. Use this object with the `fiaccel -args` and `-globals` options to specify the properties of the input arguments and the global variables, respectively. Do not pass it as an input to a generated MEX function.

You can use a `coder.Constant` object in place of a `coder.Type` object to specify a given constant value in an entry-point input or global variable.

Creation

`const_type = coder.Constant(v)` creates a `coder.Constant` type from the value `v`.

`const_type = coder.newtype('constant', v)` creates a `coder.Constant` type from the value `v`.

Note After you have created a `coder.Constant` object, you can create a constant global variable g that has the value v by using the `codegen` command: `codegen -globals {'g', coder.Constant(v)}`.

Properties

`Value` — Actual value of constant
constant

The actual value of the constant. Also indicates the input argument value `v` that is used to construct the input argument type.

Here, in the first example, when `k` is passed in `codegen` with value `v` as `42`, the corresponding input type is inferred as double. Similarly, in the second example, when `k` is passed in `codegen` with value `v` as `42`, the corresponding input type is inferred as uint8.

Example: `k = coder.Constant(42);`

Example: `k = coder.Constant(uint8(42));`

Examples

Create a Constant with Value 42

```
k = coder.Constant(42);
```

Create a new constant type for use in code generation

```
k = coder.newtype('constant', 42);
```

Limitations

- You cannot use `coder.Constant` on sparse matrices, or on structures, cell arrays, or classes that contain sparse matrices.

Version History

Introduced in R2011a

See Also

`coder.Type` | `coder.newtype` | `fiaccel`

coder.EnumType class

Package: coder
Superclasses: coder.ArrayType

Represent set of MATLAB enumerations acceptable for input specification

Description

Objects of the coder.EnumType class specify the MATLAB enumerations that the generated code accepts. Use objects of this class only with the -args option of the fiaccel command. Do not pass as an input to a generated MEX function.

Creation

t = coder.typeof(enumValue) creates a coder.EnumType object that represents a set of enumeration values of class enumValue.

t = coder.typeof(enumValue,sz,variableDims) creates a coder.EnumType type object with upper bound sizes sz and variable dimensions indicated in variableDims. If sz specifies inf for a dimension, then the size of the dimension is unbounded and the dimension has a variable size. When sz is [], the upper bound sizes of v do not change. If you do not specify variableDims, the bounded dimensions of the type are fixed and the unbounded dimensions have a variable size. When variableDims is a scalar, the function applies this value to bounded dimensions that are not 1 or 0, which are fixed.

t = coder.newtype(enumName,sz,variableDims) creates a coder.EnumType object that has a variable size with upper bound sizes sz and variable dimensions variableDims. If sz specifies inf for a dimension, then the size of the dimension is unbounded and the dimension has a variable size. If you do not specify variableDims, the bounded dimensions of the type are fixed. When variableDims is a scalar, the function applies this value to bounded dimensions that are not 1 or 0, which are fixed.

Input Arguments

enumValue — Enumeration value
enumeration object

Enumeration value defined on the MATLAB path, specified as an enumeration object.

sz — Dimensions of type object
[1 1] for coder.newtype (default) | integer vector | integer

Dimension of type object, specified as a vector of positive integers or scalar positive integer.

variableDims — Option to specify variable size
boolean vector

Option to specify whether each dimension has a variable size, specified as a boolean vector. If you specify an element of this vector as 1, the corresponding dimension has a variable size. Otherwise, the dimension has a fixed size.

enumName — Name of enumeration
string scalar | character vector

Name of enumeration defined on the MATLAB path, specified as a string scalar or character vector.

Properties

ClassName — Class of values
string scalar | character vector

Class of values in the set, returned as a string scalar or character vector.

SizeVector — Upper bound size of arrays
integer vector

Upper bound size of the arrays in the set, specified as a vector of integers.

VariableDims — Indication of whether dimensions have variable size
boolean vector

Indication of whether each dimension of the array has fixed or variable size. If a vector element is `1`, the corresponding dimension has variable size. Otherwise, the dimension has a fixed-size.

Examples

Create Enumeration Type Object

On the MATLAB® path, define an enumeration named `MyColors`.

```
type MyColors.m
```

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

Create a `coder.EnumType` object from this enumeration by using `coder.typeof`.

```
t = coder.newtype('MyColors')
```

```
t =
coder.EnumType
   1×1 MyColors

    Edit Type Object
```

Version History

Introduced in R2011a

See Also

`coder.ClassType` | `coder.Type` | `coder.ArrayType` | `coder.typeof` | `coder.newtype` | `coder.resize` | `fiaccel`

Topics

"Enumerations"
"Create and Edit Input Types by Using the Coder Type Editor"

coder.extrinsic

Declare a function as extrinsic and execute it in MATLAB

Syntax

```
coder.extrinsic(function)
coder.extrinsic(function1, ... ,functionN)

coder.extrinsic('-sync:on', function1, ... ,functionN)
coder.extrinsic('-sync:off', function1, ... ,functionN)
```

Description

`coder.extrinsic(function)` declares `function` as an extrinsic function. The code generator does not produce code for the body of the extrinsic function and instead uses the MATLAB engine to execute the call. This functionality is available only when the MATLAB engine is available during execution. Examples of situations where the MATLAB engine is available include execution of MEX functions, Simulink simulations, or function calls at the time of code generation (also known as compile time).

During standalone code generation, the code generator attempts to determine whether an extrinsic function only has a side effect (for example, by displaying a plot) or whether it affects the output of the function in which it is called (for example, by returning a value to an output variable). If there is no change to the output, the code generator proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, the code generator produces a compilation error.

You cannot use `coder.ceval` on functions that you declare as extrinsic by using `coder.extrinsic`. Also, the `coder.extrinsic` directive is ignored outside of code generation.

See “Use MATLAB Engine to Execute a Function Call in Generated Code”.

Note The code generator automatically treats many common MATLAB visualization functions, such as `plot`, `disp`, and `figure`, as extrinsic. You do not have to explicitly declare them as extrinsic functions by using `coder.extrinsic`.

`coder.extrinsic(function1, ... ,functionN)` declares `function1` through `functionN` as extrinsic functions.

`coder.extrinsic('-sync:on', function1, ... ,functionN)` enables synchronization of global data between MATLAB execution and generated code execution or Simulink simulation before and after calls to the extrinsic functions `function1` through `functionN`. If only a few extrinsic calls use or modify global data, turn off synchronization before and after all extrinsic function calls by setting the global synchronization mode to `At MEX-function entry and exit`. Use the `'-sync:on'` option to turn on synchronization for only the extrinsic calls that do modify global data.

If you use MATLAB Coder to generate a MEX function, the `'-sync:on'` option enables verification of consistency of constant global data between MATLAB and MEX functions after calls to the extrinsic functions.

See “Generate Code for Global Data” (MATLAB Coder).

`coder.extrinsic('-sync:off', function1, ... ,functionN)` disables synchronization of global data between MATLAB execution and generated code execution before and after calls to the extrinsic functions `function1` through `functionN`. If most extrinsic calls use or modify global data, but a few do not, use the `'-sync:off'` option to turn off synchronization for the extrinsic calls that do not modify global data.

If you use MATLAB Coder to generate a MEX function, the `'-sync:off'` option disables verification of consistency of constant global data between MATLAB and MEX functions after calls to the extrinsic functions.

See “Generate Code for Global Data” (MATLAB Coder).

Examples

Declare a Function as Extrinsic

The MATLAB function `str2num` is not supported for code generation. This example shows how you can still use the functionality of `str2num` in your generated MEX function by declaring `str2num` as extrinsic in your MATLAB function.

This MATLAB code declares `str2num` as extrinsic in the local function `convertStringToNumber`. By declaring `str2num` as extrinsic, you instruct the code generator not to produce code for `str2num`. Instead, the code generator dispatches `str2num` to MATLAB for execution.

```
function n = convertStringToNumber(c)
    %#codegen
    coder.extrinsic('str2num');
    n = str2num(c);
end
```

Generate a MEX function for `convertStringToNumber`.

```
codegen convertStringToNumber -args {coder.typeof('c', [1 Inf])} -report
```

In the report, you can view the generated code.

```
/* Function Definitions */
static const mxArray *str2num(const emlrtStack *sp,
                const mxArray *m1,emlrtMCInfo *location)
{
  const mxArray *m;
  const mxArray *pArray;
  pArray = m1;
  return emlrtCallMATLABR2012b((emlrtConstCTX)sp, 1, &m, 1,
                          &pArray, "str2num", true, location);
}
```

Return Output of Extrinsic Function to MATLAB at Run Time

The output that an extrinsic function returns at run time is an `mxArray`, also known as a MATLAB array. The only valid operations for an `mxArray` are storing it in a variable, passing it to another extrinsic function, or returning it to MATLAB. To perform any other opeation on an `mxArray` value, such as using it in an expression in your code, you must convert the `mxArray` to a known type at run time. To perform this action, assign the `mxArray` to a variable whose type is already defined by a prior assignment.

This example shows how to return an `mxArray` output from an extrinsic function directly to MATLAB. The next example shows how to convert the same `mxArray` output to a known type, and then use it in an expression inside your MATLAB function.

Define Entry-Point Function

Define a MATLAB function `return_extrinsic_output` that accepts source and target node indices for a directed graph as inputs and determines if the graph is acyclic by using the `hascycles` function. The `hascycles` function is not supported for code generation and is declared as extrinsic.

```
type return_extrinsic_output.m
```

```
function hasCycles = return_extrinsic_output(source,target)
coder.extrinsic('hascycles');
assert(numel(source) == numel(target))
G = digraph(source,target);
hasCycles = hascycles(G);
end
```

Generate and Call MEX Function

Generate MEX code for `return_extrinsic_output`. Specify the inputs to be unbounded vectors of type `double`.

```
codegen return_extrinsic_output -args {coder.typeof(0,[1 Inf]),coder.typeof(0,[1 Inf])} -report
```

```
Code generation successful: To view the report, open('codegen\mex\return_extrinsic_output\html\re
```

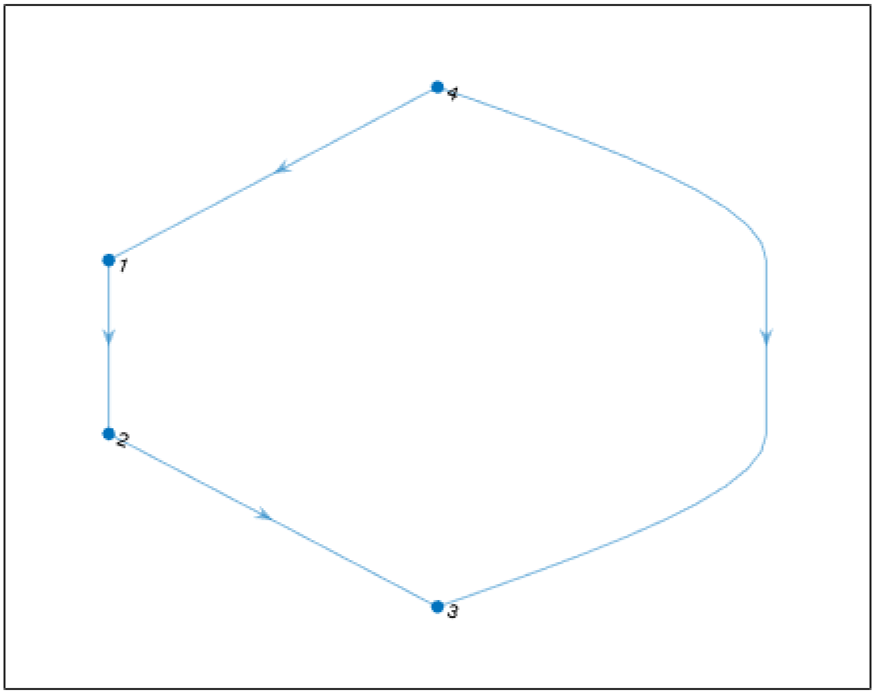
Call the generated MEX function `return_extrinsic_output_mex` with suitable inputs:

```
return_extrinsic_output([1 2 4 4],[2 3 3 1])
```

```
ans = logical
   0
```

To visually inspect if the directed graph has cycles, plot the directed graph in MATLAB.

```
plot(digraph([1 2 4 4],[2 3 3 1]))
```



Use Output of Extrinsic Function in an Expression at Run Time

The output that an extrinsic function returns is an mxArray, also known as a MATLAB array. The only valid operations for an mxArray are storing it in a variable, passing it to another extrinsic function, or returning it to MATLAB. To perform any other operation on an mxArray value, such as using it in an expression in your code, convert the mxArray to a known type at run time. To perform this action, assign the mxArray to a variable whose type is already defined by a prior assignment.

This example shows how to convert the mxArray output of an extrinsic function to a known type, and then use the output in an expression inside your MATLAB function.

Define Entry-Point Function

Define a MATLAB function use_extrinsic_output that accepts source and target node indices for a directed graph as inputs and determines if the graph is acyclic by using the hascycles function. The hascycles function is not supported for code generation and is declared as extrinsic. The entry-point function displays a message based on the output of the hascycles function.

```
type use_extrinsic_output
```

```
function use_extrinsic_output(source,target) %#codegen
assert(numel(source) == numel(target))
G = digraph(source,target);

coder.extrinsic('hascycles');
```

```
hasCycles = true;

hasCycles = hascycles(G);
if hasCycles == true
    disp('The graph has cycles')
else
    disp('The graph does not have cycles')
end
end
```

The local variable `hasCycles` is first preassigned the Boolean value `true` before the assignment `hasCycles = hascycles(G)` occurs. This preassignment enables the code generator to convert the `mxArray` that the extrinsic function `hascycles` returns to a Bsoolean before assigning it to the `hasCycles` variable. This conversion in turn enables you to compare `hasCycles` with the Boolean `true` in the condition of the `if` statement.

Generate and Call MEX Function

Generate MEX code for `use_extrinsic_output`. Specify the inputs to be unbounded vectors of type double.

```
codegen use_extrinsic_output -args {coder.typeof(0,[1 Inf]),coder.typeof(0,[1 Inf])} -report
```

```
Code generation successful: To view the report, open('codegen\mex\use_extrinsic_output\html\repor
```

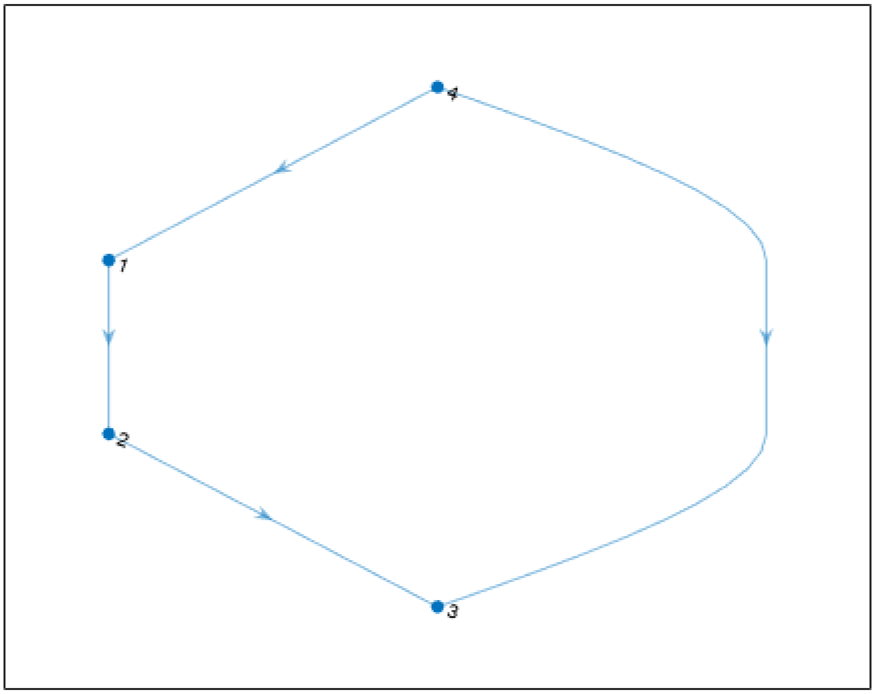
Call the generated MEX function `use_extrinsic_output_mex` with suitable inputs:

```
use_extrinsic_output_mex([1 2 4 4],[2 3 3 1])
```

```
The graph does not have cycles
```

To see if the directed graph has cycles, plot the graph in MATLAB.

```
plot(digraph([1 2 4 4],[2 3 3 1]))
```



Evaluate Extrinsic Function Call at Compile Time by Using `coder.const`

This example shows how to call an extrinsic function at the time of code generation (also known as compile time) by using `coder.const`. Because the MATLAB engine is always available during the evaluation of the expression inside `coder.const`, you can use this coding pattern when generating either MEX or standalone code. Unlike the previous two examples that show run-time execution, you do not need to explicitly convert the output of the extrinsic function to a known type if its evaluation happens at compile time.

In this example, the entry-point function `rotate_complex` invokes another function `xml2struct` that uses the MATLAB API for XML processing. Because code generation does not support the MATLAB API for XML processing, the `xml2struct` function is declared as extrinsic in the body of the entry-point function. Also, the call to `xml2struct` inside the entry-point function returns a compile-time constant. So, this output is constant-folded by placing the function call inside the `coder.const` directive.

Inspect XML File Containing Parameters

The supporting file `complex.xml` contains the values of real and imaginary parts of a complex number.

```
type complex.xml
```

```
<params>
    <param name="real" value="3"/>
```

```
    <param name="imaginary" value="4"/>
</params>
```

Define `xml2struct` Function

The MATLAB function `xml2struct` reads an XML file that uses the format of `complex.xml` to store parameter names and values, stores this information as structure fields, and returns this structure.

```
type xml2struct.m
```

```
function s = xml2struct(file)
s = struct();
import matlab.io.xml.dom.*
doc = parseFile(Parser,file);
els = doc.getElementsByTagName("params");
for i = 0:els.getLength-1
    it = els.item(i);
    ps = it.getElementsByTagName("param");
    for j = 0:ps.getLength-1
        param = ps.item(j);
        paramName = char(param.getAttribute("name"));
        paramValue = char(param.getAttribute("value"));
        paramValue = evalin("base", paramValue);
        s.(paramName) = paramValue;
    end
end
```

Define Entry-Point Function

Your MATLAB entry-point function `rotate_complex` first calls `xml2struct` to read the file `complex.xml`. It then rotates the complex number by an angle that is equal to the input argument `theta` in degrees and returns the resulting complex number.

```
type rotate_complex.m
```

```
function y = rotate_complex(theta) %#codegen
coder.extrinsic("xml2struct");
s = coder.const(xml2struct("complex.xml"));

comp = s.real + 1i * s.imaginary;
magnitude = abs(comp);
phase = angle(comp) + deg2rad(theta);
y = magnitude * cos(phase) + 1i * sin(phase);

end
```

The `xml2struct` function is declared as extrinsic and its output is constant-folded by placing the function inside the `coder.const` directive.

Generate and Inspect Static Library

Generate a static library for `read_complex` by using the `codegen` (MATLAB Coder) command. Specify the input type to be a scalar double.

```
codegen -config:lib rotate_complex -args {0} -report
```

```
Warning: Code generation is using a coder.EmbeddedCodeConfig object. Because
Embedded Coder is not installed, this might cause some Embedded Coder features
```

```
to fail.

Code generation successful (with warnings): To view the report, open('codegen\lib\rotate_complex\
```

Inspect the generated C++ file `rotate_complex.c`. Observe that the output of the `xml2struct` function is hardcoded in the generated code.

```
type codegen/lib/rotate_complex/rotate_complex.c
```

```
/*
 * File: rotate_complex.c
 *
 * MATLAB Coder version            : 5.6
 * C/C++ source code generated on  : 03-Mar-2023 07:28:32
 */

/* Include Files */
#include "rotate_complex.h"
#include <math.h>

/* Function Definitions */
/*
 * Arguments    : double theta
 * Return Type  : creal_T
 */
creal_T rotate_complex(double theta)
{
  creal_T y;
  double y_tmp;
  y_tmp = 0.017453292519943295 * theta + 0.92729521800161219;
  y.re = 5.0 * cos(y_tmp);
  y.im = sin(y_tmp);
  return y;
}

/*
 * File trailer for rotate_complex.c
 *
 * [EOF]
 */
```

Input Arguments

`function` — MATLAB function name
character vector

Name of the MATLAB function that is declared as extrinsic.

Example: `coder.extrinsic('patch')`

Data Types: `char`

Limitations

- Extrinsic function calls have some overhead that can affect performance. Input data that is passed in an extrinsic function call must be provided to MATLAB, which requires making a copy of the data. If the function has any output data, this data must be transferred back into the MEX function environment, which also requires a copy.

- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.

Tips

- The code generator automatically treats many common MATLAB visualization functions, such as `plot`, `disp`, and `figure`, as extrinsic. You do not have to explicitly declare them as extrinsic functions by using `coder.extrinsic`.
- Use the `coder.screener` function to detect which functions you must declare as extrinsic. This function runs the Code Generation Readiness Tool that screens the MATLAB code for features and functions that are not supported for code generation.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`coder.screener`

Topics

"Use MATLAB Engine to Execute a Function Call in Generated Code"
"Generate Code for Global Data" (MATLAB Coder)
"Resolution of Function Calls for Code Generation"

coder.FiType class

Package: `coder`
Superclasses: `coder.ArrayType`

Represent set of MATLAB fixed-point arrays acceptable for input specification

Description

Objects of `coder.FiType` specify the fixed-point array values that the generated code accepts. Use objects of this class only with the `-args` options of the `fiaccel` command. Do not pass as an input to the generated MEX function.

Creation

`t = coder.typeof(v)` creates a `coder.FiType` object representing a set of fixed-point values whose properties are the same as the fixed-point input `v`.

`t = coder.typeof(v,sz, variableDims)` creates a `coder.FiType` object with upper bound sizes specified by `sz` and variable dimensions indicated in `variableDims`. If `sz` specifies `Inf` for a dimension, then the size of the dimension is unbounded and variable size. When `sz` is `[]`, the upper bound sizes of `v` do not change. If you do not specify the `variableDims`, the bounded dimensions of the type are fixed. When `variableDims` is a scalar, this function applies this value to the bounded dimensions that are not `1` or `0`, which are fixed.

`t = coder.newtype('embedded.fi',numerictype,sz,variableDims)` creates a `coder.FiType` object representing a set of fixed-point values with `numerictype` and upper bound sizes `sz` and variable dimensions indicated in `variableDims`. If `sz` specifies `Inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. If you do not specify the `variableDims`, the bounded dimensions of the type are fixed. When `variableDims` is a scalar, this function applies this value to the bounded dimensions that are not `1` or `0`, which are fixed.

`t = coder.newtype('embedded.fi',numerictype,sz,variableDims,Name=Value)` creates a `coder.FiType` object representing a set of fixed-point values with additional options specified by one or more name-value pair arguments. `Name` is a property name and `Value` is the corresponding value. You can specify several name-value arguments in any order as `Name1=Value1,…,NameN=ValueN`.

Note You can create and edit `coder.Type` objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

Input Arguments

v — Input variable
fixed-point numeric value

Input variable, specified as a fixed-point numeric value.

sz — Size of type object dimensions
integer vector

Size of type object dimensions, specified as a vector of integers.

variableDims — Option to specify variable size
boolean vector

Option to specify whether each dimension has a variable size, specified as a boolean vector. If you specify an element of this vector as 1, the corresponding dimension has a variable size. Otherwise, the dimension has a fixed size.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

complex — Option to represent complex values
falseor 0 (default) | trueor 1

Option to create a coder.FiType object that can represent complex values, specified as a numeric or logical 1 (true) or 0 (false).

fimath — Fixed-point math option
fimath object

Fixed-point math option, specified as a fimath object. If you do not specify this input, the coder.FiType object uses a fimath with default property values.

Properties

ClassName — Value class name
string scalar | character vector

Value class name, returned as a string scalar.

Complex — Indication of whether fixed-point arrays are complex
0 | 1

Indication of whether the fixed-point arrays in the set are real or complex.

Fimath — Fixed-point math option
fimath object

Fixed-point math option that the fixed-point arrays in the set use, returned as a fimath object.

NumericType — Fixed-point representation option
numerictype object

Fixed-point representation option that the fixed-point arrays in the set use, returned as a numerictype object.

`SizeVector` — Upper bound size of arrays
integer vector

Upper-bound size of the arrays in the set, returned as vector of integers.

`VariableDims` — Option to specify variable-size
boolean vector

Option to specify whether each dimension of the array has a fixed or variable size. A value of `1` indicates that the corresponding element has a variable size. A value of `0` indicates that the corresponding element has a fixed size.

Examples

Create Fixed-Point Type Object

Create the fixed-point type `t`.

```
t = coder.typeof(fi(1))
```

```
t =
coder.FiType
   1×1 embedded.fi
             DataTypeMode: Fixed-point: binary point scaling
               Signedness: Signed
               WordLength: 16
           FractionLength: 14


    Edit Type Object
```

Create a fixed-point type for use in code generation. The fixed-point type uses the default `fimath` object.

```
t = coder.newtype('embedded.fi',numerictype(1, 16, 15), [1 2])
```

```
t =
coder.FiType
   1×2 embedded.fi
             DataTypeMode: Fixed-point: binary point scaling
               Signedness: Signed
               WordLength: 16
           FractionLength: 15


    Edit Type Object
```

Version History

Introduced in R2011a

See Also

`coder.ClassType` | `coder.Type` | `coder.ArrayType` | `coder.typeof` | `coder.resize` | `coder.newtype` | `fiaccel`

Topics

"Create and Edit Input Types by Using the Coder Type Editor"

coder.FixPtConfig class

Package: `coder`

Floating-point to fixed-point conversion configuration object

Description

A `coder.FixPtConfig` object contains the configuration parameters that the `fiaccel` function requires to convert floating-point MATLAB code to fixed-point MATLAB code. Use the `-float2fixed` option to pass this object to the `fiaccel` function.

Creation

`fixptcfg = coder.config('fixpt')` creates a `coder.FixPtConfig` object for floating-point to fixed-point conversion.

Properties

`ComputeDerivedRanges` — Enable derived range analysis
`false` (default) | `true`

Enable derived range analysis, specified as `true` or `false`.

`ComputeSimulationRanges` — Enable collection and reporting of simulation range data
`true` (default) | `false`

Enable collection and reporting of simulation range data, specified as `true` or `false`. If you need to run a long simulation to cover the complete dynamic range of your design, consider disabling simulation range collection and running derived range analysis instead.

`DefaultFractionLength` — Default fixed-point fraction length
4 (default) | positive integer

Default fixed-point fraction length, specified as a positive integer.

`DefaultSignedness` — Default signedness of variables in the generated code
`'Automatic'` (default) | `'Signed'` | `'Unsigned'`

Default signedness of variables in the generated code, specified as `'Automatic'`, `'Signed'`, or `'Unsigned'`.

`DefaultWordLength` — Default fixed-point word length
14 (default) | positive integer

Default fixed-point word length, specified as a positive integer.

`DetectFixptOverflows` — Enable detection of overflows using scaled doubles
`false` (default) | `true`

Enable detection of overflows using scaled doubles, specified as `true` or `false`.

`fimath` — `fimath` properties to use for conversion
`fimath('RoundingMethod','Floor','OverflowAction','Wrap','ProductMode','FullPrecision','SumMode','FullPrecision')` (default) | `fimath` object

`fimath` properties to use for conversion, specified as a `fimath` object.

`FixPtFileNameSuffix` — Suffix for fixed-point file names
`'_fixpt'` (default) | string

Suffix for fixed-point file names, specified as a string.

`LaunchNumericTypesReport` — View the numeric types report after the software has proposed fixed-point types
`true` (default) | `false`

View the numeric types report after the software has proposed fixed-point types, specified as `true` or `false`.

`LogIOForComparisonPlotting` — Enable simulation data logging to plot the data differences introduced by fixed-point conversion
`true` (default) | `false`

Enable simulation data logging to plot the data differences introduced by fixed-point conversion, specified as `true` or `false`.

`OptimizeWholeNumber` — Optimize the word lengths of variables that are always whole numbers
`true` (default) | `false`

Optimize the word lengths of variables whose simulation min/max logs indicate that they are always whole numbers, specified as `true` or `false`.

`PlotFunction` — Name of function to use for comparison plots
`''` (default) | string

Name of function to use for comparison plots.

`LogIOForComparisonPlotting` must be set to true to enable comparison plotting. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function should accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

`PlotWithSimulationDataInspector` — Use the Simulation Data Inspector for comparison plots
`false` (default) | `true`

Use the Simulation Data Inspector for comparison plots, specified as `true` or `false`.

`LogIOForComparisonPlotting` must be set to true to enable comparison plotting. The `PlotFunction` option takes precedence over `PlotWithSimulationDataInspector`.

`ProposeFractionLengthsForDefaultWordLength` — Propose fixed-point types based on `DefaultWordLength`
`true` (default) | `false`

Propose fixed-point types based on `DefaultWordLength`, specified as `true` or `false`.

`ProposeTargetContainerTypes` — Whether to propose target container types
`false` (default) | `true`

By default (`false`), propose data types with the minimum word length needed to represent the value. When set to true, propose data type with the smallest word length that can represent the range and is suitable for C code generation (8, 16, 32, 64, …). For example, for a variable with range [0..7], propose a word length of 8 rather than 3.

`ProposeWordLengthsForDefaultFractionLength` — Propose fixed-point types based on `DefaultFractionLength`
`false` (default) | `true`

Propose fixed-point types based on `DefaultFractionLength`, specified as `true` or `false`.

`ProposeTypesUsing` — Propose data types based on simulation range data, derived ranges, or both
`'BothSimulationAndDerivedRanges'` (default) | `'SimulationRanges'` | `'DerivedRanges'`

Propose data types based on simulation range data, derived ranges, or both, specified as `'BothSimulationAndDerivedRanges'`, `'SimulationRanges'`, or `'DerivedRanges'`.

`SafetyMargin` — Safety margin percentage by which to increase the simulation range when proposing fixed-point types
`0` (default) | real number greater than `-100`

Safety margin percentage by which to increase the simulation range when proposing fixed-point types, specified as a real number greater than `-100`.

Data Types: `double`

`StaticAnalysisQuickMode` — Perform faster static analysis
`false` (default) | `true`

Perform faster static analysis, specified as `true` or `false`.

`StaticAnalysisTimeoutMinutes` — Abort analysis if timeout is reached
`''` (default) | positive integer

Abort analysis if timeout is reached, specified as a positive integer.

`TestBenchName` — Test bench function name or names
`''` (default) | string | cell array of strings

Test bench function name or names, specified as a string or cell array of strings. You must specify at least one test bench. If you do not explicitly specify input parameter data types, the conversion uses the first test bench function to infer these data types.

Data Types: `string` | `cell`

`TestNumerics` — Enable numerics testing
`false` (default) | `true`

Enable numerics testing, specified as `true` or `false`.

Examples

Convert Floating-Point MATLAB Code to Fixed Point Based On Simulation Ranges

Create a `coder.FixPtConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`. The conversion process uses the test bench to infer input data types and collect simulation range data.

```
fixptcfg.TestBenchName = 'dti_test';
```

Select to propose data types based on simulation ranges only. By default, proposed types are based on both simulation and derived ranges.

```
fixptcfg.ProposeTypesUsing = 'SimulationRanges';
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccel -float2fixed fixptcfg dti
```

Convert Floating-Point MATLAB Code to Fixed Point Based On Simulation and Derived Ranges

Create a `coder.FixPtConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the name of the test bench to use to infer input data types. In this example, the test bench function name is `dti_test`. The conversion process uses the test bench to infer input data types.

```
fixptcfg.TestBenchName = 'dti_test';
```

Select to propose data types based on derived ranges.

```
fixptcfg.ProposeTypesUsing = 'DerivedRanges';
fixptcfg.ComputeDerivedRanges = true;
```

Add design ranges. In this example, the `dti` function has one scalar double input, `u_in`. Set the design minimum value for `u_in` to `-1` and the design maximum to `1`.

```
fixptcfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
```

Convert the floating-point MATLAB function, `dti`, to fixed-point MATLAB code.

```
fiaccel -float2fixed fixptcfg  dti
```

Enable Overflow Detection

When you select to detect potential overflows, `fiaccel` generates a scaled double version of the generated fixed-point MEX function. Scaled doubles store their data in double-precision floating-point, so they carry out arithmetic in full range. They also retain their fixed-point settings, so they are able to report when a computation goes out of the range of the fixed-point type.

Create a `coder.FixPtConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Enable numerics testing with overflow detection.

```
fixptcfg.TestNumerics = true;
fixptcfg.DetectFixptOverflows = true;
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccel -float2fixed fixptcfg dti
```

Alternatives

You can convert floating-point MATLAB code to fixed-point code using the **Fixed-Point Converter** app. Open the app using one of these methods:

- On the **Apps** tab, in the **Code Generation** section, select **Fixed-Point Converter**.
- Use the `fixedPointConverter` command.

Version History

Introduced in R2014b

See Also

`fiaccel` | `coder.mexconfig`

Topics

“Propose Data Types Based on Simulation Ranges”
“Propose Data Types Based on Derived Ranges”
“Detect Overflows”

coder.ignoreConst

Prevent use of constant value of expression for function specializations

Syntax

`coder.ignoreConst(expression)`

Description

`coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations on page 4-161. `coder.ignoreConst(expression)` returns the value of `expression`.

Examples

Prevent Function Specializations Based on Constant Input Values

Use `coder.ignoreConst` to prevent function specializations for a function that is called with constant values.

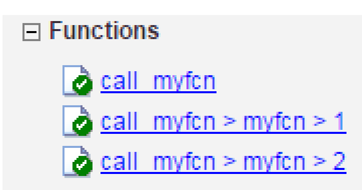
Write the function `call_myfn`, which calls `myfcn`.

```
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, 'mode1');
y = myfcn(n, 'mode2');
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode,'mode1')
    y = n;
else
    y = -n;
end
end
```

Generate standalone C code. For example, generate a static library. Enable the code generation report.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you see two function specializations for `call_myfcn`.



The code generator creates `call_myfcn>myfcn>1` for `mode` with a value of `'mode1'`. It creates `call_myfcn>myfcn>2` for `mode` with a value of `'mode2'`.

In the generated C code, you see the specializations `my_fcn` and `b_my_fcn`.

```
static double b_myfcn(double n)
{
  return -n;
}

static double myfcn(double n)
{
  return n;
}
```

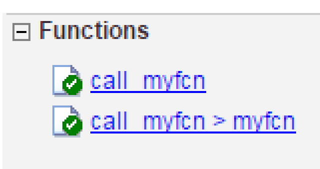
To prevent the function specializations, instruct the code generator to ignore that values of the `mode` argument are constant.

```
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, coder.ignoreConst('mode1'));
y = myfcn(n, coder.ignoreConst('mode2'));
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode,'mode1')
    y = n;
else
    y = -n;
end
end
```

Generate the C code.

```
codegen -config:lib call_myfcn -args {1} -report
```

In the code generation report, you do not see multiple function specializations.



In the generated C code, you see one function for `my_fcn`.

Input Arguments

`expression` — Expression whose value is to be treated as a nonconstant
MATLAB expression

Expression whose value is to be treated as a nonconstant, specified as a MATLAB expression.

More About

Function Specialization

Version of a function in which an input type, size, complexity, or value is customized for a particular invocation of the function.

Function specialization produces efficient C code at the expense of code duplication. The code generation report shows all MATLAB function specializations that the code generator creates. However, the specializations might not appear in the generated C/C++ code due to later transformations or optimizations.

Tips

- For some recursive function calls, you can use `coder.ignoreConst` to force run-time recursion. See “Force Code Generator to Use Run-Time Recursion”.
- `coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations. It does not prevent other uses of the constant value during code generation.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also
`coder.inline`

Topics
“Force Code Generator to Use Run-Time Recursion”
“Compile-Time Recursion Limit Reached”

coder.inline

Package: `coder`

Control inlining of a specific function in generated code

Syntax

```
coder.inline('always')
coder.inline('never')
coder.inline('default')
```

Description

`coder.inline('always')` forces inlining on page 4-163 of the current function in the generated code. Place the `coder.inline` directive inside the function that you want to inline. The code generator does not inline entry-point functions and recursive functions. Also, the code generator does not inline functions into `parfor` loops, or inline functions called from `parfor` loops.

`coder.inline('never')` prevents inlining of the current function in the generated code. Prevent inlining when you want to simplify the mapping between the MATLAB source code and the generated code.

Note If you use the `codegen` or the `fiaccel` command, you can disable inlining for all functions by using the `-O disable:inline` option.

If you generate C/C++ code by using the `codegen` command or the MATLAB Coder app, you might have different speed and readability requirements for the code generated for functions that you write and the code generated for MathWorks® functions. Certain additional global settings enable you to separately control the inlining behavior for these two parts of the generated code base and at the boundary between them. See “Control Inlining to Fine-Tune Performance and Readability of Generated Code” (MATLAB Coder).

In cases where the code generator does not prevent inlining of a function in the generated code even if it contains the `coder.inline('never')` directive, use the `coder.ignoreConst` function on an input at the function call site in your MATLAB code. For more information, see “Resolve Issue: coder.inline('never') Does Not Prevent Inlining of Function” (MATLAB Coder).

`coder.inline('default')` instructs the code generator to use internal heuristics to determine whether to inline the current function. Usually, the heuristics produce highly optimized code. Use `coder.inline` explicitly in your MATLAB functions only when you need to fine-tune these optimizations.

Examples

Prevent Function Inlining

In this example, function `foo` is not inlined in the generated code:

```
function y = foo(x)
  coder.inline('never');
  y = x;
end
```

Use coder.inline in Control Flow Statements

You can use `coder.inline` in control flow code. If the software detects contradictory `coder.inline` directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose that you want to generate code for a division function that runs on a system with limited memory. To optimize memory use in the generated code, the `inline_division` function manually controls inlining based on whether it performs scalar division or vector division:

```
function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if isscalar(dividend) && isscalar(divisor)
   coder.inline('always');
else
% Vector division produces a for-loop.
% Prohibit inlining to reduce code size.
   coder.inline('never');
end

if any(divisor == 0)
   error('Cannot divide by 0');
end

y = dividend / divisor;
```

More About

Inlining

Technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`fiaccel`

Topics

“Resolve Issue: coder.inline('never') Does Not Prevent Inlining of Function” (MATLAB Coder)

coder.load

Load compile-time constants from MAT-file or ASCII file

Syntax

```
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat',var1,...,varN)
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

Description

`S = coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`coder.load` loads data at code generation time, also referred to as *compile time*. If you change the content of `filename` after you generate code, the change is not reflected in the behavior of the generated code.

`S = coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

`S = coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

`S = coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.

`S = coder.load(filename,'-mat')` treats `filename` as a MAT-file, regardless of the file extension.

`S = coder.load(filename,'-mat',var1,...,varN)` treats `filename` as a MAT-file and loads only the specified variables from the file.

`S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)` treats `filename` as a MAT-file and loads only the variables that match the specified regular expressions.

Examples

Load compile-time constants from MAT-file

Generate code for a function edgeDetect1 which given a normalized image, returns an image where the edges are detected with respect to the threshold value. edgeDetect1 uses coder.load to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];

save sobel.mat k
```

Write the function edgeDetect1.

```
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat','k');
H = conv2(double(originalImage),S.k, 'same');
V = conv2(double(originalImage),S.k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for edgeDetect1.

```
codegen -report -config cfg edgeDetect1
```

codegen generates C code in the codegen\lib\edgeDetect1 folder.

Load compile-time constants from ASCII file

Generate code for a function edgeDetect2 which given a normalized image, returns an image where the edges are detected with respect to the threshold value. edgeDetect2 uses coder.load to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

```
k = [1 2 1; 0 0 0; -1 -2 -1];
save sobel.dat k -ascii
```

Write the function edgeDetect2.

```
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

k = coder.load('sobel.dat');
H = conv2(double(originalImage),k, 'same');
V = conv2(double(originalImage),k','same');
```

```
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```

Create a code generation configuration object for a static library.

```
cfg = coder.config('lib');
```

Generate a static library for `edgeDetect2`.

```
codegen -report -config cfg edgeDetect2
```

`codegen` generates C code in the `codegen\lib\edgeDetect2` folder.

Input Arguments

`filename` — Name of file
character vector | string scalar

Name of file. `filename` must be a compile-time constant.

`filename` can include a file extension and a full or partial path. If `filename` has no extension, `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, `%`).

Example: `'myFile.mat'`

`var1,...,varN` — Names of variables to load
character vector | string scalar

Names of variables, specified as one or more character vectors or string scalars. Each variable name must be a compile-time constant. Use the `*` wildcard to match patterns.

Example: `coder.load('myFile.mat','A*')` loads all variables in the file whose names start with `A`.

`expr1,...,exprN` — Regular expressions indicating which variables to load
character vector | string scalar

Regular expressions indicating which variables to load specified as one or more character vectors or string scalars. Each regular expression must be a compile-time constant.

Example: `coder.load('myFile.mat', '-regexp', '^A')` loads only variables whose names begin with `A`.

Output Arguments

`S` — Loaded variables or data
structure array | m-by-n array

If `filename` is a MAT-file, `S` is a structure array.

If `filename` is an ASCII file, `S` is an m-by-n array of type `double`. m is the number of lines in the file and n is the number of values on a line.

Limitations

- Arguments to `coder.load` must be compile-time constants.
- The output `S` must be the name of a structure or array without any subscripting. For example, `S(i) = coder.load('myFile.mat')` is not allowed.
- You cannot use `save` to save workspace data to a file inside a function intended for code generation. The code generator does not support the `save` function. Furthermore, you cannot use `coder.extrinsic` with `save`. Prior to generating code, you can use `save` to save workspace data to a file.

Tips

- `coder.load(filename)` loads data at compile time, not at run time. If you change the content of `filename` after you generate code, the change is not reflected in the behavior of the generated code. If you are generating MEX code or code for Simulink simulation, you can use the MATLAB function `load` to load run-time values.
- If the MAT-file contains unsupported constructs, use `coder.load(filename,var1,...,varN)` to load only the supported constructs.
- If you generate code in a MATLAB Coder project, the code generator practices incremental code generation for the `coder.load` function. When the MAT-file or ASCII file used by `coder.load` changes, the software rebuilds the code.

Version History

Introduced in R2013a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`matfile` | `regexp` | `save`

Topics

"Regular Expressions"

coder.newtype

Package: `coder`

Create `coder.Type` object to represent type of an entry-point function input

Syntax

```
t = coder.newtype(numeric_class,sz,variable_dims)
t = coder.newtype(numeric_class,sz,variable_dims, Name,Value)
t = coder.newtype('constant',value)
t = coder.newtype('struct',struct_fields,sz,variable_dims)
t = coder.newtype('cell',cells,sz,variable_dims)
t = coder.newtype('embedded.fi',numerictype,sz,variable_dims, Name,Value)
t = coder.newtype(enum_value,sz,variable_dims)
t = coder.newtype('class_name')
t = coder.newtype('string')
```

Description

The `coder.newtype` function is an advanced function that you can use to control the `coder.Type` object. Consider using `coder.typeof` instead of `coder.newtype`. The function `coder.typeof` creates a type from a MATLAB example. By default, `t = coder.newtype('class_name')` does not assign any properties of the class, `class_name` to the object `t`.

Note You can also create and edit `coder.Type` objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

`t = coder.newtype(numeric_class,sz,variable_dims)` creates a `coder.Type` object representing values of class `numeric_class`, sizes `sz` (upper bound), and variable dimensions `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable-size. When `variable_dims` is not specified, the dimensions of the type are fixed except for those that are unbounded. When `variable_dims` is a scalar, it is applied to type dimensions that are not `1` or `0`, which are fixed.

`t = coder.newtype(numeric_class,sz,variable_dims, Name,Value)` creates a `coder.Type` object by using additional options specified as one or more Name, Value pair arguments.

`t = coder.newtype('constant',value)` creates a `coder.Constant` object representing a single value. Use this type to specify a value that must be treated as a constant in the generated code.

`t = coder.newtype('struct',struct_fields,sz,variable_dims)` creates a `coder.StructType` object for an array of structures that has the same fields as the scalar structure `struct_fields`. The structure array type has the size specified by `sz` and variable-size dimensions specified by `variable_dims`.

`t = coder.newtype('cell',cells,sz,variable_dims)` creates a `coder.CellType` object for a cell array that has the cells and cell types specified by `cells`. The cell array type has the size specified by `sz` and variable-size dimensions specified by `variable_dims`. You cannot change the number of cells or specify variable-size dimensions for a heterogeneous cell array.

`t = coder.newtype('embedded.fi',numerictype,sz,variable_dims, Name,Value)` creates a `coder.FiType` object representing a set of fixed-point values that have `numerictype` and additional options specified by one or more Name, Value pair arguments.

`t = coder.newtype(enum_value,sz,variable_dims)` creates a `coder.Type` object representing a set of enumeration values of class `enum_value`.

`t = coder.newtype('class_name')` creates a `coder.ClassType` object for an object of the class `class_name`. The new object does not have any properties of the class `class_name`.

`t = coder.newtype('string')` creates a `coder.StringType` object for a string scalar. A string scalar contains one piece of text represented as a character vector. To specify the size of the character vector and whether the second dimension is variable-size, set the `StringLength` property to the required size and set `VariableStringLength` to `true`. For example, `t.StringLength = 10` and `t.VariableStringLength = true` specifies that the string scalar is variable-size with an upper bound of 10.

Examples

Create Type for a Matrix

Create a type for a variable-size matrix of doubles.

```
t = coder.newtype('double',[2 3 4],[1 1 0])
```

```
t =

coder.PrimitiveType
   :2×:3×4 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, and second dimension with fixed size.

```
t = coder.newtype('double',[inf,3])
```

```
t =

coder.PrimitiveType
   :inf×3 double
```

```
t = coder.newtype('double',[inf,3],[1 0])
```

```
%  also returns
t =

coder.PrimitiveType
   :inf×3 double
%  ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, and second dimension with variable-size that has an upper bound of 3.

```
t = coder.newtype('double',[inf,3],[0 1])
```

```
t = 

coder.PrimitiveType
   :inf×:3 double

%  ':' indicates variable-size dimensions
```

Create Type for a Structure

Create a type for a structure with a variable-size field.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
t = coder.newtype('struct',struct('a',ta,'b',tb),[1 1],[1 1])
```

```
t = 

coder.StructType
   :1×:1 struct
      a: 1×1 int8
      b: :1×:2 double
% ':' indicates variable-size dimensions
```

Create Type for a Cell Array

Create a type for a heterogeneous cell array.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
t = coder.newtype('cell',{ta, tb})
```

```
t = 

coder.CellType
   1×2 heterogeneous cell
      f1: 1×1 int8
      f2: :1×:2 double
% ':' indicates variable-size dimensions
```

Create a type for a homogeneous cell array.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('int8',[1 2],[1 1]);
t = coder.newtype('cell',{ta, tb},[1,1],[1,1])
```

```
t = 

coder.CellType
   :1×:1 homogeneous cell
      base: :1×:2 int8
% ':' indicates variable-size dimensions
```

Create Type for a Constant

Create a new constant type to use in code generation.

```
t = coder.newtype('constant',42)
```

```
t = 

coder.Constant
       42
```

Create a coder.EnumType Object

Create a `coder.EnumType` object by using the name of an existing MATLAB enumeration.

1. Define an enumeration `MyColors`. On the MATLAB path, create a file named MyColors containing:

```
classdef MyColors < int32
    enumeration
        green(1),
        red(2),
    end
end
```

2. Create a `coder.EnumType` object from this enumeration.

```
t = coder.newtype('MyColors')
```

```
t = 

coder.EnumType
   1×1 MyColors
```

Create a Fixed-Point Type

Create a fixed-point type for use in code generation.

The fixed-point type uses default fimath values.

```
t = coder.newtype('embedded.fi',numerictype(1, 16, 15),[1 2])
```

```
t = 

coder.FiType
   1×2 embedded.fi
             DataTypeMode: Fixed-point: binary point scaling
               Signedness: Signed
               WordLength: 16
           FractionLength: 15
```

Create a Type for an Object

Create a type for an object to use in code generation.

1. Create this value class:

```
classdef mySquare
    properties
        side;
    end

    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
            end
        end

        function a = calcarea(obj)
            a = obj.side * obj.side;
        end

    end
end
```

2. Create a type for an object that has the same properties as `mySquare`.

```
t = coder.newtype('mySquare');
```

3. The previous step creates a `coder.ClassType` type for `t`, but does not assign any properties of `mySquare` to it. To ensure `t` has all the properties of `mySquare`, change the type of the property `side` by using `t.Properties`.

```
t.Properties.side = coder.typeof(int8(3))
```

```
t =

coder.ClassType
   1×1 mySquare
      side: 1×1 int8
```

Create Type for a String Scalar

Create a type for a string scalar to use in code generation.

1. Create the string scalar type.

```
t = coder.newtype('string');
```

2. Specify the size.

```
t.StringLength = 10;
```

3. Make the string variable-size.

```
t.VariableStringLength = true;
```

4. To make the string variable-size with no upper bound, set `StringLength` to `Inf`.

```
t.StringLength = Inf;
```

Note Setting `StringLength` to `Inf` implicitly sets `VariableStringLength` to `true`.

Input Arguments

`numeric_class` — Class of values of type object
numeric (default)

Class of the set of values represented by the type object.

Example: `coder.newtype('double',[6,3]);`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

`struct_fields` — Indicates fields in a new structure type
struct (default)

Scalar structure used to specify the fields in a new structure type.

Example: `coder.newtype('struct',struct('a',ta,'b',tb));`

Data Types: `struct`

`cells` — Specify types of cells in a new cell array type
cell array (default)

Cell array of `coder.Type` objects that specify the types of the cells in a new cell array type.

Example: `coder.newtype('cell',{ta,tb});`

Data Types: `cell`

`sz` — Dimension of type object
row vector of integer values

Size vector specifying each dimension of type object. The `sz` dimension cannot change the number of cells for a heterogeneous cell array.

Example: `coder.newtype('int8',[1 2]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

`'class_name'` — Name of the class
character vector | string scalar

Name of the class from which the `coder.ClassType` is created. Specify as a character vector or string scalar. `class_name` must be the name of a value class.

Example: `coder.newtype('mySquare')`

Data Types: `char` | `string`

`variable_dims` — Variable- or fixed-dimension
row vector of logical values

The value of `variable_dims` is `true` for dimensions for which `sz` specifies an upper bound of `inf`; `false` for all other dimensions.

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false). You cannot specify variable-size dimensions for a heterogeneous cell array.

Example: `coder.newtype('char',[1,10],[0,1]);`

Data Types: `logical`

`value` — Value of the constant
constant value (default)

Specifies the actual value of the constant.

Example: `coder.newtype('constant',41);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell`

`enum_value` — Enumeration values of class
enum (default)

Enumeration values of a class.

Example: `coder.newtype('MyColors');`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `t = coder.newtype('double',[10 20],sparse = true)`

`complex` — Type representing complex values
`false` (default) | `true`

Set `complex` to `true` to create a `coder.Type` object that can represent complex values. The type must support complex data.

`fimath` — Type representing `fimath` values
`fimath` object with default property settings | `fimath` object

Specify local `fimath`. If you do not specify `fimath`, the code generator uses default `fimath` values.

Use this option only when you create a `coder.FiType` object.

`sparse` — Type representing sparse data
`false` (default) | `true`

Set `sparse` to `true` to create a `coder.Type` object representing sparse data. The type must support sparse data.

This option is not supported when:

- You create a `coder.FiType` object.
- You set the `gpu` option to `true`.

gpu — Type representing GPU inputs
`false` (default) | `true`

Set `gpu` to `true` to create a `coder.Type` object that can represent the GPU input type. This option requires GPU Coder™.

Output Arguments

t — New type object
coder.Type object

A new `coder.Type` object.

Limitations

- For sparse matrices, `coder.newtype` drops upper bounds for variable-size dimensions.
- For GPU input types, only bounded numeric and logical base types are supported. Scalar GPU arrays, structures, cell-arrays, classes, enumerated types, character, half-precision and fixed-point data types are not supported.
- When using `coder.newtype` to represent GPU inputs, the memory allocation (malloc) mode property of the GPU code configuration object to `'discrete'`.

Tips

- The `coder.newtype` function fixes the size of a singleton dimension unless the `variable_dims` argument explicitly specifies that the singleton dimension has a variable size.

  For example, this code specifies a 1-by-:10 double. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

  ```
  t = coder.newtype('double',[1 10],1)
  ```

  By contrast, this code specifies a :1-by-:10 double. Both dimensions have a variable size.

  ```
  t = coder.newtype('double',[1 10],[1 1])
  ```
- For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

Alternatives

`coder.typeof`

Version History

Introduced in R2011a

See Also

`coder.resize` | `coder.Type` | `coder.ArrayType` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.CellType` | `fiaccel` | `coder.OutputType`

Topics

"Create and Edit Input Types by Using the Coder Type Editor"

coder.nullcopy

Package: `coder`

Declare uninitialized variables in code generation

Syntax

`X = coder.nullcopy(A)`

Description

`X = coder.nullcopy(A)` copies type, size, and complexity of `A` to `X`, but does not copy element values. The function preallocates memory for `X` without incurring the overhead of initializing memory. In code generation, the `coder.nullcopy` function declares uninitialized variables. In MATLAB, `coder.nullcopy` returns the input such that X is equal to A.

If X is a structure or a class containing variable-sized arrays, then you must assign the size of each array. `coder.nullcopy` does not copy sizes of arrays or nested arrays from its argument to its result.

Note Before you use `X` in a function or a program, ensure that the data in `X` is completely initialized. Declaring a variable through `coder.nullcopy` without assigning all the elements of the variable results in nondeterministic program behavior. For more information, see "How to Eliminate Redundant Copies by Defining Uninitialized Variables".

Examples

Declare a Variable Without Initialization

This example shows how to declare an array type variable without initializing any value in the array.

To generate code for the following function, you must fully declare the output variable `outp` as a *n*-by-*n* array of real doubles before subscripting into `outp`. To perform this declaration without initializing all the values in the array, use `coder.nullcopy`.

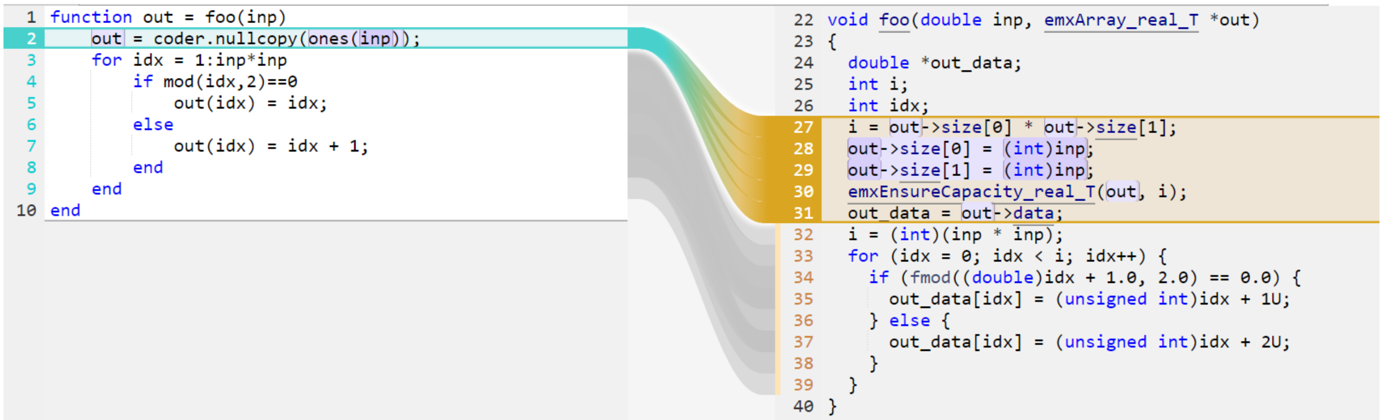
```
function outp = foo(n) %#codegen

outp = coder.nullcopy(ones(n));
for idx = 1:n*n
   if mod(idx,2) == 0
      outp(idx) = idx;
   else
      outp(idx) = idx + 1;
   end
end
```

Run this `codegen` command to generate code and launch report.

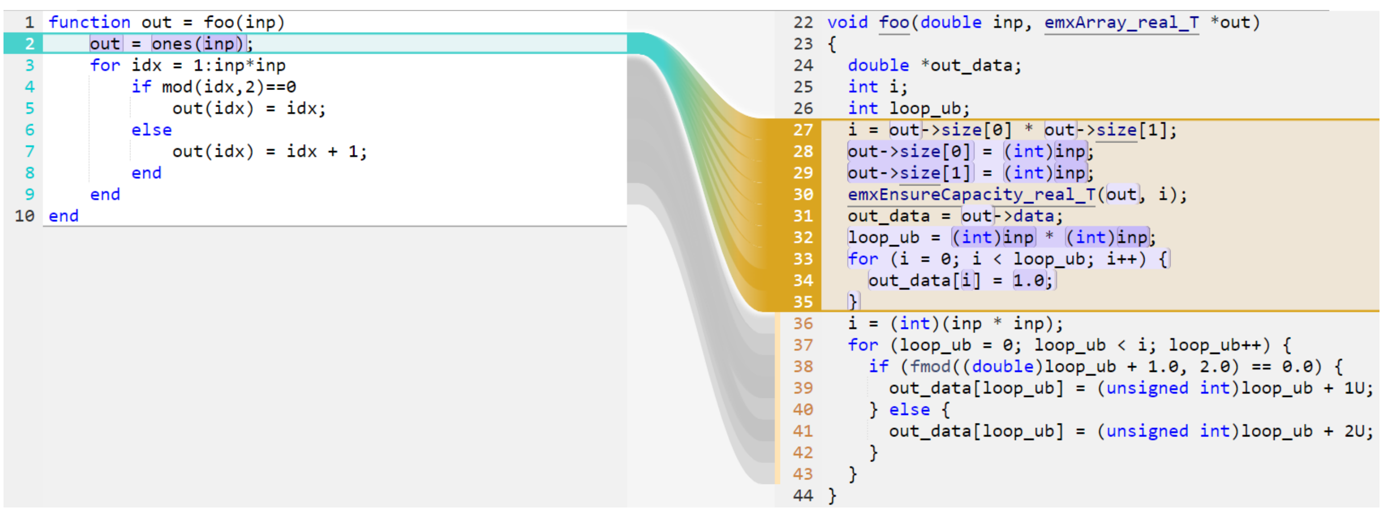
```
codegen -config:lib -c foo -args {0} -launchreport
```

In the code generation report, click **Trace Code** to see the mapping between the MATLAB code and the generated code. To use the code traceability feature, you must have Embedded Coder®.

The following figures show the comparison between the code generated with and without `coder.nullcopy`. Using `coder.nullcopy` with `ones` can specify the size of array `outp` without initializing each element to one.



If you do not use `coder.nullcopy`, the generated code explicitly initializes every element in `outp` to one (see lines 32 to 35).



Note In some situations, the code generator automatically performs the optimization corresponding to `coder.nullcopy`, even if you do not explicitly include the `coder.nullcopy` directive in your MATLAB code.

Input Arguments

A — Variable to copy

scalar | vector | matrix | class | multidimensional array

Variable to copy, specified as a scalar, vector, matrix, or multidimensional array.

Example: `coder.nullcopy(A);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `class`
Complex Number Support: Yes

Limitations

- You cannot use `coder.nullcopy` on sparse matrices.
- You cannot use `coder.nullcopy` with classes that support overloaded parentheses or require indexing methods to access their data, such as `table`.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Topics
"Eliminate Redundant Copies of Variables in Generated Code"

coder.read

Package: `coder`

Read data files at run time in generated code

Syntax

```
dataFromFile = coder.read(fileName)
dataFromFile = coder.read(fileName,TypeHeaderFrom=typeHeaderFilename)
[dataFromFile,errID] = coder.read( ___ )
```

Description

`dataFromFile = coder.read(fileName)` reads from the `fileName.coderdata` storage file and returns the data stored within the file. This syntax works for a constant `fileName` input only. Use this function in your MATLAB code for which you want to generate C/C++ code. The generated code performs the data read at run time.

To store your workspace variables in a `.coderdata` file, use the `coder.write` function in MATLAB.

`dataFromFile = coder.read(fileName,TypeHeaderFrom=typeHeaderFilename)` uses the information contained in `typeHeaderFilename` to determine the type and size of the data to be read from `fileName`. Both `fileName` and `typeHeaderFilename` must be `.coderdata` files. The `typeHeaderFilename` argument must be a compile-time constant and the file that this name represents must exist in your current directory during code generation.

Each `.coderdata` file contains a type header that specifies the type and size of the data stored in the file. The code generated for the `coder.read` function can read any `.coderdata` file at run-time, while the file type and size is consistent with the type and size information that you supply using the `typeHeaderFilename` file during code generation.

To create a `.coderdata`file to use with the `TypeHeaderFrom` argument, use the `coder.write` function in MATLAB.

`[dataFromFile,errID] = coder.read( ___ )` suppresses run-time errors during a read operation. `errID` is a `coder.ReadStatus` enumeration object. If any errors occur, `coder.read` returns the first error through `errID` and `dataFromFile` returns unusable content. Use this option to test the generated code for targets for which run-time errors are disabled.

Examples

Read Data from a Fixed `.coderdata` File at Run Time

In this example, you use `coder.write` to create a `.coderdata` file that stores a single array from your MATLAB workspace. You then generate code for a `coder.read` function call that reads this file at run time.

Create a 20-by-20 array of double type in your workspace.

```
c = rand(20);
```

Store this variable in a file named `exampleData.coderdata` in your current directory.

```
coder.write("exampleData.coderdata",c);
```

To read from a `.coderdata` file with the constant file name `exampleData`, use the `coder.read` function as shown in your MATLAB entry-point function.

```
function data = readSingleFile %#codegen
data = coder.read("exampleData.coderdata");
end
```

Generate a MEX function for `readSingleFile`.

```
codegen readSingleFile -report
```

Read the data stored in `exampleData.coderdata` at run time by running the generated MEX.

```
readSingleFile_mex
```

Read Data from Multiple .coderdata Files at Run Time

This example shows how to generate code for a `coder.read` command that can read multiple `.coderdata` files at run time. These files contain array data that have the same type, but different sizes. To enable a single `coder.read` call to read all these files, pass a type header file that is consistent with all your individual data files to the `coder.read` function call.

To start, create a storage file that you want the generated code to read. The following `coder.write` command creates the storage file `file_a.coderdata`, which contains two variables with array data of type double. Variables `a` and `b` are of different sizes.

```
a = rand(10,20);
b = rand(5,30);
coder.write("file_a.coderdata",a);
```

A `coder.Type` object that is consistent with both variables `a` and `b` must have variable-size dimensions. The upper bounds of the two array dimensions must be at least 10 and 30 respectively. Create a `coder.Type` object that represents a variable-size double type with these bounds.

```
t = coder.typeof(a,[10 30],[1 1])
```

```
t = 

coder.PrimitiveType
   :10×:30 double
```

You can modify the header information of `file_a.coderdata` and use the modified file as the source of the type header information.

```
coder.write("file_a.coderdata",a,TypeHeader=t);
```

You can also create the `file_b.coderdata` file with the required type header information by running this command.

```
coder.write("file_b.coderdata",b,TypeHeader=t);
```

Create a MATLAB entry-point function, `readMultipleFiles`, that can read `file_a.coderdata` and `file_b.coderdata`.

```
function data = readMultipleFiles(filename) %#codegen
data = coder.read(filename,TypeHeaderFrom="file_b.coderdata");
end
```

Generate a MEX for `readMultipleFiles`. Specify the input argument type as an unbounded variable-size character vector.

```
codegen readMultipleFiles -args {coder.typeof('a',[1 inf])} -report
```

Run the generated MEX with inputs `'file_a.coderdata'` and `'file_b.coderdata'`.

```
readMultipleFiles_mex('file_a.coderdata')
readMultipleFiles_mex('file_b.coderdata')
```

Input Arguments

`fileName` — Name of `.coderdata` storage file

string scalar (default) | character vector

Name of the `.coderdata` storage file from which you want to read data, specified as a string scalar or character vector.

To store your workspace variables in a `.coderdata` file, use the `coder.write` function in MATLAB.

`typeHeaderFilename` — Constant name of `.coderdata` file that stores type and size information

string scalar | character vector

Constant file name of a `.coderdata` file that stores type and size information about the files to read at run time, specified as a string scalar or character vector.

Each `.coderdata` file contains a type header that specifies the type and size of the data stored in the file. The code generated for the `coder.read` function can read any `.coderdata` file at run time while the file type and size is consistent with the type and size information that you supply using the `typeHeaderFilename` file during code generation. This file is also referred to as the type header file.

To create a type header file, use the `coder.write` function in MATLAB.

Output Arguments

`dataFromFile` — Data read from `.coderdata` storage file

array | structure | cell array | enumeration | categorical | sparse array

Data read from the `.coderdata` storage file, returned as an array or multiple arrays stored within a structure or cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `categorical` | `sparse`

`errID` — Read error enumeration object

`coder.ReadStatus` object

Read error enumeration object, specified as one of these values.

Enumeration Member	Enumeration Value	Error Reference
`Success`	0	Read operation success.
`CoderReadCouldNotOpen`	1	Unable to open specified `.coderdata` file.
`CoderReadProblemReading`	2	Issue while reading `.coderdata` file.
`CoderReadUnexpectedValue`	3	Unexpected value in `.coderdata` file.
`CoderReadWrongHeader`	4	`.coderdata` file does not contain expected metadata. The input file might be corrupted or is not a `.coderdata` file. Use `coder.write` to create `.coderdata` files.
`CoderReadWrongVersion`	5	`.coderdata` file is not compatible with this release of MATLAB Coder. Create a new `.coderdata` file with this version of the product to generate a compatible file.
`CoderReadStructArray`	6	Expected to read a scalar structure, but `.coderdata` file contains a structure array. Use a compatible `TypeHeader` to read this file or a constant file name.

`MATFile`	7	`coder.read` cannot read `MAT` file. Convert your `MAT` file to a `.coderdata` files by running these commands in the command window:
		`s = load('MATFileName');`
		`coder.write('fileName.coderdata',s);`
		Read the new `.coderdata` file by using `coder.read`.

`WrongType`	8	The type information of the `.coderdata` file does not match the type information in the specified by the `TypeHeaderFrom` argument.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`coder.write` | `coder.load` | `fread` | `fwrite` | `fscanf` | `fprintf`

Topics
"Data Read and Write Considerations" (MATLAB Coder)

coder.write

Package: `coder`

Create data files that the generated code reads at run time

Syntax

```
coder.write(fileName,data)
coder.write(fileName,data,Name=Value)
```

Description

`coder.write(fileName,data)` stores the argument `data` in a file with the name `fileName.coderdata` in your current directory. Use this function in MATLAB execution only and not the MATLAB code you intend to use for C/C++ code generation.

Use the `coder.read` function to read data from a `.coderdata` file in MATLAB and in the generated code.

A `.coderdata` file contains a type header that specifies the type and size of the data stored in the file. The code generated for the `coder.read` function can read any `.coderdata` file at run-time, whose type and size is consistent with the type and size information that you supply to `coder.read` during code generation.

`coder.write(fileName,data,Name=Value)` creates a type header file, which is a `.coderdata` file that you use to specify the type and size of data to the `coder.read` function. Use the name-value arguments to:

- Specify a custom type header for the `.coderdata` file that is consistent with the variable `data`.
- Omit the actual data and create a `.coderdata` file with a type header only.

Examples

Store Variable in `.coderdata` File

This example shows how to create a storage file for a single variable. This storage file can be read at run-time by using the `coder.read` function.

Create a 20-by-20 array of double type in your workspace.

```
c = rand(20);
```

Store this variable in a file named `storageFile.coderdata` in your current directory.

```
coder.write("storageFile.coderdata",c);
```

Store Multiple Variables in .coderdata File

This example shows how to create a storage file for multiple variables.

Create two arrays of double type in your workspace. Array a is of size 10-by-10 and array b is 20-by-20.

```
a = rand(10);
b = rand(20);
```

Create structure s that contains arrays a and b in its fields.

```
s = struct('a',a,'b',b);
```

Store the structure s in a .coderdata file.

```
coder.write("storageFile.coderdata",s);
```

Alternatively, you can use a cell array to store multiple variables in a .coderdata file.

Create Type Header File Compatible with Multiple Data Files

In this example, you generate code for a coder.read command that can read multiple .coderdata files at run time. These files contain array data that have the same type, but different sizes. To generate code, you must pass a type header file that is consistent with all your individual data files to the coder.read function call.

Create the storage file file_a.coderdata, which contains two variables with array data of type double. Variables a and b are of different sizes.

```
a = rand(10,20);
b = rand(5,30);
coder.write("file_a.coderdata",a);
```

A coder.Type object that is consistent with both variables a and b must have variable-size dimensions. The upper bounds of the two array dimensions must be at least 10 and 30 respectively. Create a coder.Type object that represents a variable-size double type with these bounds.

```
t = coder.typeof(a,[10 30],[1 1])
```

```
t =

coder.PrimitiveType
   :10×:30 double
```

You can modify the header information of file_a.coderdata and use the modified file as the source of the type header information.

```
coder.write("file_a.coderdata",a,TypeHeader=t);
```

You can also create the file_b.coderdata file with the required type header information by running this command.

```
coder.write("file_b.coderdata",b,TypeHeader=t);
```

Alternatively, you can create a separate type header file, `myTypeHeader.coderdata`, that contains only the type header compatible with all the existing storage files and does not contain any actual data.

```
coder.write("file_b.coderdata",b,TypeHeader=t,TypeHeaderOnly=true);
```

Such a type header file is useful if you are working with large data files, and want to use the generated file as a type header file only.

Input Arguments

`fileName` — Name or full file path of new or existing storage file
string scalar | character vector

Name or full file path of a new or existing storage file, specified as a string scalar or character vector. The generated storage file has the name `fileName.coderdata`, or just `fileName` if the value you supplied already has the extension.

`data` — Data source to save in storage file
array | structure | cell array | enumeration | categorical | sparse array

Data source to save in the storage file, specified as an array or multiple arrays stored within a structure or cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `cell` | `categorical` | `sparse`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `coder.write('example.coderdata',rand(1000),TypeHeader=coder.typeof(1,[1000 1000],1 1]),TypeHeaderOnly=true)`

`TypeHeader` — Type information for storage data
`coder.typeof(data)` (default) | `coder.Type` object

Type information for storage data, specified as a `coder.Type` object. This type object must be compatible with the input argument `data`.

To create a type object, use the `coder.typeof` or `coder.newtype` function. You can also create and edit `coder.Type` objects interactively by using the Coder Type Editor. See “Create and Edit Input Types by Using the Coder Type Editor” (MATLAB Coder).

Example: `coder.write('example.coderdata',rand(10),TypeHeader=coder.typeof(1,[10 10],1 1]))`

`TypeHeaderOnly` — Whether to create `.coderdata` file with type header only
`false` (default) | `true`

Option to specify whether to omit the data and create a `.coderdata` file with the type header only, specified as `true` or `false`. If `data` is a large array and you want to use the `.coderdata` file as a type header file only, set this argument to `true`.

Example: `coder.write('example.coderdata',rand(1000),TypeHeaderOnly=true)`

`Verbose` — Whether to report when writing a file
`true` (default) | `false`

Option to specify whether to report when writing a `.coderdata` file at the MATLAB command line, specified as `true` or `false`.

Version History

Introduced in R2023a

See Also

`coder.read` | `coder.load` | `fread` | `fwrite` | `fscanf` | `fprintf`

Topics

"Data Read and Write Considerations" (MATLAB Coder)

coder.PrimitiveType class

Package: `coder`
Superclasses: `coder.ArrayType`

Represent set of logical, numeric, or character arrays acceptable for input specification

Description

Objects of `coder.PrimitiveType` specify logical, numeric, or character values that the generated code accepts. Supported types are `half`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `char`, and `logical`. Use objects of this class only with the `-args` option of the `fiaccel` command. Do not pass as an input to a generated MEX function.

Creation

`t = coder.typeof(v)` creates a `coder.PrimitiveType` object denoting the smallest non-constant type that contains `v`. `v` must be a MATLAB numeric, logical or character.

`t = coder.typeof(v,sz,variableDims)` creates a `coder.PrimitiveType` object with upper bound sizes specified by `sz` and variable dimensions indicated in `variableDims`. If `sz` specifies `Inf` for a dimension, then the size of the dimension is unbounded and variable sized. When `sz` is `[]`, the upper bound sizes of `v` remain unchanged. If you do not specify the `variableDims`, the bounded dimensions of the type are fixed. When `variableDims` is a scalar, this function applies this value to the bounded dimensions that are not `1` or `0`, which are fixed.

`t = coder.newtype(numericClass,sz,variableDims)` creates a `coder.PrimitiveType` object representing values of class `numericClass` with upper bound sizes `sz` and variable dimensions indicated in `variableDims`. If `sz` specifies `Inf` for a dimension, then the size of the dimension is unbounded and variable sized. If you do not specify the `variableDims`, the bounded dimensions of the type are fixed. When `variableDims` is a scalar, this function applies this value to the bounded dimensions that are not `1` or `0`, which are fixed.

`t = coder.newtype(numericClass,sz,variableDims,Name=Value)` creates a `coder.PrimitiveType` object with additional options specified by one or more name,-value arguments. `Name` is a property name and `Value` is the corresponding value. Specify `Name` as character vector or string scalar. You can specify several name-value arguments in any order as `Name1=Value1,…,NameN=ValueN`.

Note You can create and edit `coder.Type` objects interactively by using the Coder Type Editor. See “Create and Edit Input Types by Using the Coder Type Editor”.

Input Arguments

v — Input variable
numeric | logical | character

Input variable, specified as a numeric, logical, or character value.

`sz` — Size of type object dimensions
integer vector

Size of type object dimensions, specified as a vector of integers.

`variableDims` — Option to specify variable size
boolean vector

Option to specify whether each dimension has a variable size, specified as a boolean vector. If you specify an element of this vector as `1`, the corresponding dimension has a variable size. Otherwise, the dimension has a fixed size.

`numericClass` — Primitive class type
string scalar | character vector

Primitive class type, specified as a string scalar or character vector.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`complex` — Option to represent complex values
`false`or `0` (default) | `true`or `1`

Option to create a `coder.PrimitiveType` object that can represent complex values, specified as a numeric or logical `1` (`true`) or `0` (`false`).

`sparse` — Option to represent sparse data
`false`or `0` | `true` or `1`

Option to create a `coder.PrimitiveType` object that can represent sparse values, specified as a numeric or logical `1` (`true`) or `0` (`false`).

`gpu` — Option to represent GPU data
`false`or `0` | `true` or `1`

Option to create a `coder.PrimitiveType` object that can represent GPU data values, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Properties

`complex` — Option to represent complex values
`1` (default) | `0`

Option to represent complex values, specified as a `0` or `1`. The type must support complex data. Character arrays do not support complex data.

`sparse` — Option to represent sparse data
`1` | `0`

Option to represent sparse data, specified as a `0` or `1`. The type must support sparse data. Character and half-precision data types do not support sparse data.

gpu — Option to represent GPU data
`1` | `0`

Option to represent the GPU input type, specified as a `0` or `1`. This option requires a GPU Coder license. The type must support GPU data. Character and half-precision data types do not support GPU arrays.

Examples

Create Primitive Type Object

Use `coder.typeof` and specify the input variable, dimensions, and variable-size flag.

```
z = coder.typeof(0,[2 3 4],[1 1 0])
```

```
z =
coder.PrimitiveType
   :2×:3×4 double

    Edit Type Object
```

Use Primitive Type Object for Code Generation

Create a `coder.PrimitiveType` object.

```
z = coder.typeof(0,[2 3 4],[1 1 0])
```

Generate a C library for a MATLAB function that has one input parameter of type `z`.

```
codegen -config:lib fcn -args {z}
```

Version History

Introduced in R2011a

See Also

`coder.ClassType` | `coder.Type` | `coder.ArrayType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `fiaccel`

Topics
"Create and Edit Input Types by Using the Coder Type Editor"

coder.resize

Package: `coder`

Resize `coder.Type` object

Syntax

```
t_out = coder.resize(t,sz)
t_out = coder.resize(t,sz,variable_dims)
t_out = coder.resize(t,[],variable_dims)
t_out = coder.resize(t,sz,variable_dims,Name,Value)
t_out = coder.resize(t,'sizelimits',limits)
```

Description

`t_out = coder.resize(t,sz)` resizes `t` to have size `sz`.

`t_out = coder.resize(t,sz,variable_dims)` returns a modified copy of `coder.Type` `t` with (upper-bound) size `sz` and variable dimensions `variable_dims`. If `variable_dims` or `sz` are scalars, the function applies the scalars to all dimensions of `t`. By default, `variable_dims` does not apply to dimensions where `sz` is `0` or `1`, which are fixed. Use the 'uniform' option to override this special case. The `coder.resize` function ignores `variable_dims` for dimensions with size `inf`. These dimensions are variable size. `t` can be a cell array of types, in which case, `coder.resize` resizes all elements of the cell array.

`t_out = coder.resize(t,[],variable_dims)` changes `t` to have variable dimensions `variable_dims` while leaving the size unchanged.

`t_out = coder.resize(t,sz,variable_dims,Name,Value)` resizes `t` by using additional options specified by one or more Name, Value pair arguments.

`t_out = coder.resize(t,'sizelimits',limits)` resizes the individual dimensions of `t` based on the threshold values in the `limits` vector. The `limits` vector is a row vector containing two positive integer elements. Each dimension of `t` is individually resized according to the thresholds in the `limits` vector.

- When the size `S` of a dimension is lesser than both thresholds defined in `limits`, the dimension remains the same.
- When the size `S` of a dimension is greater than or equal to the first threshold and less than the second threshold defined in `limits`, the dimension becomes variable size with upper bound `S`.
- However, when the size `S` of a dimension is also greater than or equal to the second threshold defined in `limits`, the dimension becomes an unbounded variable size.

If the value of `limits` is scalar, the threshold gets scalar-expanded to represent both thresholds. For example, if `limits` is defined as `4`, it is interpreted as `[4 4]`.

The `'sizelimits'` option allows you to dynamically allocate memory to large arrays in your generated code.

Examples

Change Fixed-Size Array to an Unbounded, Variable-Size Array

Change a fixed-size array to an unbounded, variable-size array.

```
t = coder.typeof(ones(3,3))
```

```
t = 

coder.PrimitiveType
   3×3 double
```

```
coder.resize(t,inf)
```

```
ans = 

coder.PrimitiveType
   :inf×:inf double
% ':' indicates variable-size dimensions
```

Change Fixed-Size Array to a Bounded, Variable-Size Array

Change a fixed-size array to a bounded, variable-size array.

```
t = coder.typeof(ones(3,3))
```

```
t = 

coder.PrimitiveType
   3×3 double
```

```
coder.resize(t,[4 5],1)
```

```
ans = 

coder.PrimitiveType
   :4×:5 double
% ':' indicates variable-size dimensions
```

Resize Structure Field

Resize a structure field.

```
ts = coder.typeof(struct('a',ones(3, 3)))
```

```
ts = 

coder.StructType
   1×1 struct
      a: 3×3 double
```

```
coder.resize(ts,[5, 5],'recursive',1)
```

```
ans =

coder.StructType
   5×5 struct
      a: 5×5 double
```

Resize Cell Array

Resize a cell array.

```
tc = coder.typeof({1 2 3})
```

```
tc =

coder.CellType
   1×3 homogeneous cell
      base: 1×1 double
```

```
coder.resize(tc,[5, 5],'recursive',1)
```

```
ans =

coder.CellType
   5×5 homogeneous cell
      base: 1×1 double
```

Change Fixed-Sized Array to Variable-Size Based on Bounded and Unbounded Thresholds

Change a fixed-sized array to a variable size based on bounded and unbounded thresholds.

```
t = coder.typeof(ones(100,200))
```

```
t =

coder.PrimitiveType
   100×200 double
```

```
coder.resize(t,'sizelimits',[99 199])
```

```
ans =

coder.PrimitiveType
   :100×:inf double
% ':' indicates variable-size dimensions
```

Input Arguments

`limits` — Vector that defines the threshold

row vector of integer values

A row vector of variable-size thresholds. If the value of `limits` is scalar, the threshold gets scalar-expanded. If the size `sz` of a dimension of `t` is greater than or equal to the first threshold and less than the second threshold defined in `limits`, the dimension becomes variable size with upper bound `sz`. If the size `sz` of a dimension of `t` is also greater than or equal to the second threshold, the dimension becomes an unbounded variable size.

However, if the size `sz` is lesser than both thresholds, the dimension remains the same.

Example: `coder.resize(t,'sizelimits',[99 199]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — New size for object type
row vector of integer values

New size for `coder.Type` object, `t_out`

Example: `coder.resize(t,[3,4]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`t` — `coder.Type` object that you want to resize
coder.Type object

If `t` is a `coder.CellType` object, the `coder.CellType` object must be homogeneous.

Example: `coder.resize(t,inf);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

`variable_dims` — Variable or fixed dimension
row vector of logical values

Specify whether each dimension of `t_out` is fixed size or variable size.

Example: `coder.resize(t,[4 5],1);`

Data Types: `logical`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `coder.resize(t,[5, 5],'recursive', 1);`

`recursive` — Resize `t` and all types contained within it
false (default) | true

Setting `recursive` to `true` resizes `t` and all types contained within it.

Data Types: `logical`

`uniform` — Resize `t` by applying the heuristic for dimensions of size one
false (default) | true

Setting `uniform` to `true` resizes `t` and applies the heuristic for dimensions of size one.

The heuristic works in the following manner:

- If `variable_dims` is a scalar `true`, all dimensions are resized to upper bound variable sizes specified in `sz`. This includes dimensions of size one. For example:

  ```
  t = coder.typeof(1, [1 5]);
  tResize = coder.resize(t,[1 7],true,'uniform',true);
  ```

  This generates an object `tResize` as shown:

  ```
  tResize =

  coder.PrimitiveType
     :1×:7 double

      Edit Type Object
  ```

- If you set `uniform` to `true` with the `'sizelimits'` option, the dimensions of size one are also resized to variable size, according to the `'sizelimits'` heuristics. For example:

  ```
  t = coder.typeof(1, [1 5]);
  tResize = coder.resize(t,[],'sizelimits',[0 6],'uniform',true);
  ```

  These commands generate an object `tResize` as shown:

  ```
  tResize =

  coder.PrimitiveType
     :1×:5 double

      Edit Type Object
  ```

- If `variable_dims` is specified as a non-scalar logical, the `uniform` setting has no effect. However, if `variable_dims` is scalar and `uniform` is set to `false`, only dimensions of size greater than one are resized.

Data Types: `logical`

`sizelimits` — Resize individual dimensions of `t` according to thresholds provided in the `limits` vector
`limits` (default)

Using the `sizelimits` options with `limits` vector resizes individual dimensions of `t`.

```
t = coder.typeof(1, [1 5]);
tResize = coder.resize(t,[],'sizelimits',[0 6],'uniform',true);
```

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`t_out` — Resized type object
coder.Type object

Resized `coder.Type` object

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`

Complex Number Support: Yes

Limitations

- For sparse matrices, `coder.resize` drops the upper bounds for variable-size dimensions.

Version History

Introduced in R2011a

See Also

`coder.typeof` | `coder.newtype` | `fiaccel`

coder.screener

Package: `coder`

Determine if function is suitable for code generation

Syntax

```
coder.screener(fcn)
coder.screener(fcn,'-gpu')
coder.screener(fcn_1,...,fcn_n)
info = coder.screener( ___ )
```

Description

`coder.screener(fcn)` analyzes the entry-point MATLAB function `fcn` to identify unsupported functions and language features as code generation compliance issues. The code generation compliance issues are displayed in the readiness report.

If `fcn` calls other functions directly or indirectly that are not MathWorks functions (MATLAB built-in functions and toolbox functions), `coder.screener` analyzes these functions. It does not analyze the MathWorks functions.

It is possible that `coder.screener` does not detect all code generation issues. Under certain circumstances, it is possible that `coder.screener` reports false errors.

To avoid undetected code generation issues and false errors, before generating code, verify that your MATLAB code is suitable for code generation by performing these additional checks:

- Before using `coder.screener`, fix issues that the Code Analyzer identifies.
- After using `coder.screener`, and before generating C/C++ code, verify that your MATLAB code is suitable for code generation by generating and verifying a MEX function.

The `coder.screener` function does not report functions that the code generator treats as extrinsic. Examples of such functions are `plot`, `disp`, and `figure`. See "Use MATLAB Engine to Execute a Function Call in Generated Code".

`coder.screener(fcn,'-gpu')` analyzes the entry-point MATLAB function `fcn` to identify unsupported functions and language features for GPU code generation.

`coder.screener(fcn_1,...,fcn_n)` analyzes multiple entry-point MATLAB functions.

`info = coder.screener( ___ )` returns a `coder.ScreenerInfo` object. The properties of this object contain the code generation readiness analysis results. Use `info` to access the code generation readiness results programmatically. For a list of properties, see coder.ScreenerInfo Properties.

Examples

Identify Unsupported Functions

The `coder.screener` function identifies calls to functions that are not supported for code generation. It checks the entry-point function, `foo1`, and the function, `foo2`, that `foo1` calls.

Write the function `foo2` and save it in the file `foo2.m`.

```
function [tf1,tf2] = foo2(source,target)
G = digraph(source,target);
tf1 = hascycles(G);
tf2 = isdag(G);
end
```

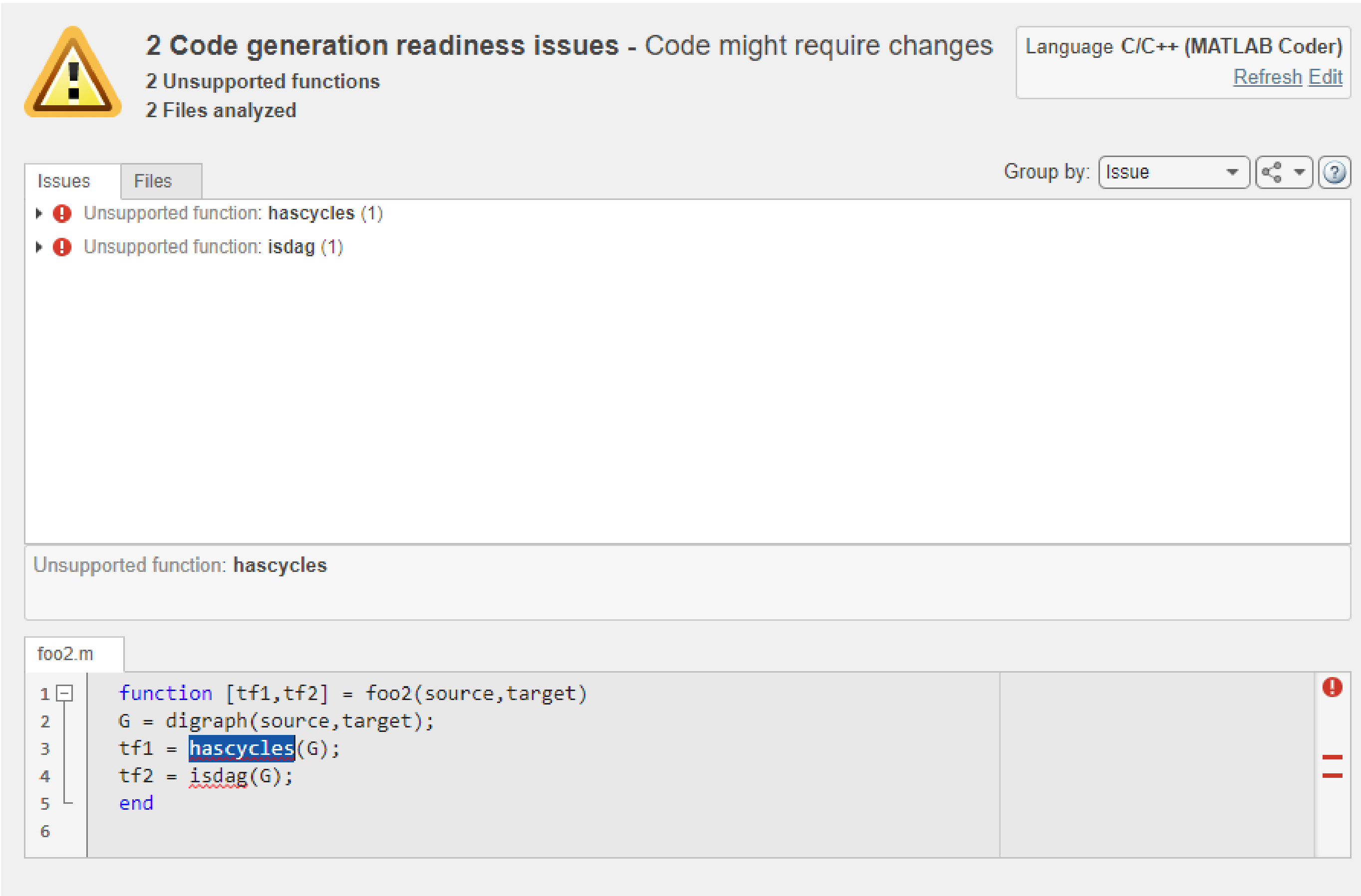
Write the function `foo1` that calls `foo2`. Save `foo1` in the file `foo1.m`.

```
function [tf1,tf2] = foo1(source,target)
assert(numel(source)==numel(target))
[tf1,tf2] = foo2(source,target);
end
```

Analyze `foo1`.

```
coder.screener('foo1')
```

The Code Generation Readiness report displays a summary of the unsupported MATLAB function calls. The report **Issues** tab indicates that `foo2.m` contains one call to the `isdag` function and one call to the `hascycles`, which are not supported for code generation.



The function `foo2` calls two unsupported MATLAB functions. To generate a MEX function, modify the code to make the calls to `hascycles` and `isdag` extrinsic by using the `coder.extrinsic` directive, and then rerun the code generation readiness tool.

```
function [tf1,tf2] = foo2(source,target)
coder.extrinsic('hascycles','isdag');
G = digraph(source,target);
tf1 = hascycles(G);
tf2 = isdag(G);
end
```

Rerun `coder.screener` on the entry-point function `foo1`.

```
coder.screener('foo1')
```

The report no longer flags that code generation does not support the `hascycles` and `isdag` functions. When you generate a MEX function for `foo1`, the code generator dispatches these two functions to MATLAB for execution.

Access Code Generation Readiness Results Programmatically

You can call the `coder.screener` function with an optional output argument. If you use this syntax, the `coder.screener` function returns a `coder.ScreenerInfo` object that contains the results of the code generation readiness analysis for your MATLAB code base. See coder.ScreenerInfo Properties.

This example uses the files `foo1.m` and `foo2.m` defined in the previous example. Call the `coder.screener` function:

```
info = coder.screener('foo1.m')
```

```
info = 

  ScreenerInfo with properties:

                Files: [2×1 coder.CodeFile]
             Messages: [2×1 coder.Message]
    UnsupportedCalls: [2×1 coder.CallSite]

    View Screener Report
```

To access information about the first unsupported call, index into the `UnsupportedCalls` property,

```
firstCall = info.UnsupportedCalls(1)
```

```
firstCall = 

  CallSite with properties:

    CalleeName: 'hascycles'
          File: [1×1 coder.CodeFile]
    StartIndex: 78
      EndIndex: 86
```

View the text of the file that contains this unsupported call to `hascycles`.

```
firstCall.File.Text
```

```
ans =

    'function [tf1,tf2] = foo2(source,target)
     G = digraph(source,target);
     tf1 = hascycles(G);
     tf2 = isdag(G);
     end
     '
```

To export the entire code generation readiness report to a MATLAB string, use the `textReport` function.

```
reportString = textReport(info)
```

```
reportString =

    'Code Generation Readiness (Text Report)
     =======================================

     2 Code generation readiness issues
     2 Unsupported functions
     2 Files analyzed

     Configuration
```

```
=============

Language: C/C++ (MATLAB Coder)

Code Generation Issues
======================

Unsupported function: digraph (2)
    - foo2.m (Line 3)
    - foo2.m (Line 4)

'
```

Identify Unsupported Data Types

The `coder.screener` function identifies MATLAB data types that code generation does not support.

Write the function `myfun1` that contains a MATLAB calendar duration array data type.

```
function out = myfun1(A)
out = calyears(A);
end
```

Analyze `myfun1`.

```
coder.screener('myfun1');
```

The code generation readiness report indicates that the `calyears` data type is not supported for code generation. Before generating code, fix the reported issue.

Input Arguments

`fcn` — Name of entry-point function
character vector | string scalar

Name of entry-point MATLAB function for analysis. Specify as a character vector or a string scalar.

Example: `coder.screener('myfun');`

Data Types: `char` | `string`

`fcn_1,...,fcn_n` — List of entry-point function names
character vector | string scalar

Comma-separated list of entry-point MATLAB function names for analysis. Specify as character vectors or string scalars.

Example: `coder.screener('myfun1','myfun2');`

Data Types: `char` | `string`

Alternatives

- "Run the Code Generation Readiness Tool From the Current Folder Browser"

Version History

Introduced in R2012b

See Also

`coder.extrinsic` | `fiaccel`

Topics

"Functions Supported for Code Acceleration or C Code Generation"
"Code Generation Readiness Tool"

coder.StructType class

Package: `coder`
Superclasses: `coder.ArrayType`

Represent set of MATLAB structure arrays acceptable for input specification

Description

Objects of `coder.StructType` specify the structure arrays that the generated code should accept. Use objects of this class only with the `-args` option of the `fiaccel` command. Do not pass as an input to a generated MEX function.

Creation

`t = coder.typeof(structV)` creates a `coder.StructType` object for a structure with the same fields as the scalar structure `struct_v`.

`t = coder.typeof(structV,sz,variableDims)` creates a `coder.StructType` with upper bound sizes specified by `sz` and variable dimensions indicated by `variableDims`. If `sz` specifies `Inf` for a dimension, then the size of the dimension unbounded and variable size. When `sz` is `[]`, the upper bound sizes of `structV` remain unchanged. If you do not specify the `variableDims`, the bounded dimensions of the type are fixed. When `variableDims` is a scalar, this function applies this value to the bounded dimensions that are not `1` or `0`, which are fixed.

`t = coder.newtype('struct',structV,sz,variableDims)` creates a `coder.StructType` object for an array of structures with the same fields as the scalar structure `structV` and upper bound size `sz` and variable dimensions indicated in `variableDims`. If `sz` specifies `Inf` for a dimension, then the size of the dimension is assumed to be unbounded and the dimension is assumed to be variable sized. If you do not specify the `variableDims`, the bounded dimensions of the type are fixed. When `variableDims` is a scalar, this function applies this value to the bounded dimensions that are not `1` or `0`, which are fixed.

Note You can create and edit `coder.Type` objects interactively by using the Coder Type Editor. See “Create and Edit Input Types by Using the Coder Type Editor”.

Input Arguments

`structV` — Input structure variable
scalar structure

Input structure variable that specifies the fields in a new structure type, specified as a scalar structure.

`sz` — Size of type object dimensions
integer vector

Size of type object dimensions, specified as a vector of integers.

`variableDims` — Option to specify variable size
boolean vector

Option to specify whether each dimension has a variable size, specified as a boolean vector. If you specify an element of this vector as `1`, the corresponding dimension has a variable size. Otherwise, the dimension has a fixed size.

Properties

`Alignment` — Run-time memory alignment
`-1` | power of `2` no greater than `128`

The run-time memory alignment of structures of this type in bytes.

If you have an Embedded Coder license and use code replacement libraries (CRLs), you can align data objects that are specified as inputs to a replacement function to a specified boundary. Use this capability to take advantage of target-specific function implementations that require data to be aligned. By default, the class does not align the structure to a specific boundary, which means that CRL functions that require alignment do not match the default structure.

`ClassName` — Value class name
character vector | string scalar

Value class name, returned as a string scalar.

`Extern` — Indication of whether structure is defined externally
`1` | `0`

Indication of whether structure is defined externally, returned as a `1` or `0`. A value of `1` indicates that the structure is defined externally. A value of `0` indicates that the structure is defined internally.

`Fields` — Field types
scalar structure

Types of fields in the structure, specified as a structure.

`HeaderFile` — External header file name
character vector

External header file name, returned as a nonempty character vector or string scalar. If the structure type is externally defined, name of the header file that contains the external definition of the structure, for example, `"mystruct.h"`.

By default, the generated code contains `#include` statements for custom header files after the standard header files. If a standard header file refers to the custom structure type, then the compilation fails. By specifying the `HeaderFile` property, MATLAB Coder includes the header file in the required location.

`SizeVector` — Upper bound of type object
integer vector | integer scalar

Upper bound of type object, specified as a vector of integer or scalar integer.

`VariableDims` — Option to specify variable-size
boolean vector

Option to specify whether each dimension of the array has a fixed or variable size. A value of `1` indicates that the corresponding element has a variable size. A value of `0` indicates that the corresponding element has a fixed size.

Examples

Create `coder.StructType` Type Object

This example shows how to create a type for a structure with a variable-size field.

Create a type object by using `coder.typeof`.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
t = coder.typeof(x)
```

```
t = 
coder.StructType
   1×1 struct
      a: :3×:5 double
      b: 3×3 double

    Edit Type Object
```

Create Externally Defined Structure Type

Create an externally defined structure type.

```
S.a = coder.typeof(double(0));
S.b = coder.typeof(single(0));
T = coder.typeof(S);
T = coder.cstructname(T,'mytype','extern',HeaderFile='myheader.h');
```

View the types of the structure fields.

```
T.Fields
```

```
ans = struct with fields:
    a: [1x1 coder.PrimitiveType]
    b: [1x1 coder.PrimitiveType]
```

Specify `coder.StructType` Object to Generate Code

Create a structure type.

```
ta = coder.newtype('int8',[1 1]);
tb = coder.newtype('double',[1 2],[1 1]);
z = coder.newtype('struct',struct('a',ta,'b',tb))
```

```
coder.StructType
   1x1 struct
      a: 1x1 int8
      b: :1x:2 double
```

Generate a C library for a MATLAB function `fcn.m` that has one input parameter of this type.

```
codegen -config:lib fcn -args {z}
```

Version History

Introduced in R2011a

See Also

`coder.ClassType` | `coder.Type` | `coder.PrimitiveType` | `coder.EnumType` | `coder.FiType` | `coder.Constant` | `coder.ArrayType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `fiaccel`

Topics

"Create and Edit Input Types by Using the Coder Type Editor"

coder.target

Determine if code generation target is specified target

Syntax

```
tf = coder.target(target)
```

Description

`tf = coder.target(target)` returns true (1) if the code generation target is `target`. Otherwise, it returns false (0).

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns true.

Examples

Use `coder.target` to Parametrize a MATLAB Function

Parametrize a MATLAB function so that it works in MATLAB or in generated code. When the function runs in MATLAB, it calls the MATLAB function `myabsval`. The generated code, however, calls a C library function `myabsval`.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)
%#codegen
y = abs(u);
```

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```
function y = callmyabsval(y)
%#codegen
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
  % Executing in MATLAB, call function myabsval
  y = myabsval(y);
else
```

```
  % add the required include statements to generated function code
  coder.updateBuildInfo('addIncludePaths','$(START_DIR)\codegen\lib\myabsval');
  coder.cinclude('myabsval_initialize.h');
  coder.cinclude('myabsval.h');
  coder.cinclude('myabsval_terminate.h');

  % Executing in the generated code.
  % Call the initialize function before calling the
  % C function for the first time
  coder.ceval('myabsval_initialize');

  % Call the generated C library function myabsval
  y = coder.ceval('myabsval',y);

  % Call the terminate function after
  % calling the C function for the last time
  coder.ceval('myabsval_terminate');
end
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use `coder.updateBuildInfo` to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsval',100,tru
```

Note The `START_DIR` macro is only supported for generating code with MATLAB Coder.

Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```
callmyabsval_mex(-2.75)
```

```
ans =

    2.7500
```

Call the MATLAB function `callmyabsval`.

```
callmyabsval(-2.75)
```

```
ans =

    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

Input Arguments

target — code generation target

'MATLAB' | 'C' | 'C++' | 'CUDA' | 'OpenCL' | 'SystemC' | 'SystemVerilog' | 'Verilog' | 'VHDL' | 'MEX' | 'Sfun' | 'Rtw' | 'HDL ' | 'Custom'

Code generation target, specified as a character vector or a string scalar. Specify one of these targets.

`'MATLAB'`	Running in MATLAB (not generating code)
`'C'`, `'C++'`, `'CUDA'`, `'OpenCL'` `'SystemC'`, `'SystemVerilog'`, `'Verilog'`, `'VHDL'`	Supported target languages for code generation
`'MEX'`	Generating a MEX function
`'Sfun'`	Simulating a Simulink model. Also used for running in Accelerator mode.
`'Rtw'`	Generating a LIB, DLL, or EXE target. Also used for running in Simulink Coder and Rapid Accelerator mode.
`'HDL'`	Generating an HDL target
`'Custom'`	Generating a custom target

Example: `tf = coder.target('MATLAB')`

Example: `tf = coder.target("MATLAB")`

Note In case of `CUDA` or `SystemC` code generation, `coder.target('C++')` is always `true`.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

coder.Type class

Package: coder

Represent set of MATLAB values acceptable for input specification

Description

Objects of coder.Type specify the values that the generated code accepts. Use objects of this class only with the -args option of the fiaccel command. Do not pass as an input to a generated MEX function.

Creation

Note You can create and edit coder.Type objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

coder.Type is an abstract class. To create objects of this class, use the coder.typeof and coder.newtype functions.

These classes are the instances of the coder.Type class.

- coder.CellType
- coder.ClassType
- coder.Constant
- coder.EnumType
- coder.FiType
- coder.OutputType
- coder.PrimitiveType
- coder.StructType

Properties

ClassName — Value class name
coder.CellType | coder.ClassType | coder.Constant | coder.EnumType | coder.FiType | coder.OutputType | coder.PrimitiveType | coder.StructType

Value class name, returned as an object of one of these classes.

- coder.CellType
- coder.ClassType
- coder.Constant
- coder.EnumType
- coder.FiType

- `coder.OutputType`
- `coder.PrimitiveType`
- `coder.StructType`

Version History
Introduced in R2011a

See Also
`fiaccel`

Topics
"Create and Edit Input Types by Using the Coder Type Editor"

coderTypeEditor

Launch the Coder Type Editor dialog

Syntax

```
coderTypeEditor
coderTypeEditor var1 ... varN
coderTypeEditor -all
coderTypeEditor -close
```

Description

`coderTypeEditor` opens an empty Coder Type Editor dialog. If a dialog is already open, this command brings it to the front of the screen.

You can use the Coder Type Editor to create and edit `coder.Type` objects interactively. See “Create and Edit Input Types by Using the Coder Type Editor”.

`coderTypeEditor var1 ... varN` opens a Coder Type Editor dialog pre-populated with `coder.Type` objects corresponding to the workspace variables `var1` through `varN`. For a variable `var`, the name of the generated `coder.Type` object is `varType`.

`coderTypeEditor -all` opens a Coder Type Editor dialog pre-populated with `coder.Type` objects corresponding to all compatible variables in the current workspace.

`coderTypeEditor -close` closes an open Coder Type Editor dialog.

Examples

Open Coder Type Editor Populated with Types for Existing Variables

In your MATLAB workspace, define variables `var1`, `var2`, and `var3`.

```
myArray = magic(4);
myCharVector = 'Hello, World!';
myStruct = struct('a',5,'b','mystring');
```

Open the type editor pre-populated with types for `var1`, `var2`, and `var3`.

```
coderTypeEditor myArray myCharVector myStruct
```

The Coder Type Editor dialog opens. The **Type Browser** pane displays the name, class (data type), and size for `coder.Type` objects `myArrayType`, `myCharVectorType`, and `myStructType` for the three workspace variables.

Inspect the created types and check that they are consistent with the variables in the workspace.

- `myArrayType` represents a 4-by-4 array of type double.
- `myCharVectorType` represents a 1-by-13 character row vector.

- `myStructType` represents a scalar of type `struct`. Expand the tree corresponding to `myStructType` in the **Type Browser**. The field `a` represents a scalar double. The field `b` represents a `1`-by-`8` character vector.

To save these types in the base workspace, in the Coder Type Editor toolstrip, click **Save**. The variables `myArrayType`, `myCharVectorType`, and `myStructType` appear in the base workspace.

Input Arguments

`var1 ... varN` — Workspace variables whose types you intend to view in the type editor
value belonging to a fundamental MATLAB class that supports code generation | value object | handle object | `coder.Type` object

Workspace variables whose types you intend to view in the type editors. They can store any value that is compatible with code generation.

The value can also be a `coder.Type` object. In that case, the `coder.Type` object itself opens in the type editor.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `categorical` | `datetime` | `duration` | `timetable` | `fi` | `value object` | `coder.Type object`
Complex Number Support: Yes

Version History

Introduced in R2020a

See Also

`coder.typeof` | `coder.newtype`

Topics
"Create and Edit Input Types by Using the Coder Type Editor"

coder.typeof

Package: coder

Create coder.Type object to represent the type of an entry-point function input

Syntax

```
type_obj = coder.typeof(v)
type_obj = coder.typeof(v,sz,variable_dims)
type_obj = coder.typeof(v,'Gpu', true)
type_obj = coder.typeof(type_obj)
```

Description

Note You can also create and edit coder.Type objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

type_obj = coder.typeof(v) creates an object that is derived from coder.Type to represent the type of v for code generation. Use coder.typeof to specify only input parameter types. For example, use it with the fiaccel function -args option. Do not use it in MATLAB code from which you intend to generate a MEX function.

type_obj = coder.typeof(v,sz,variable_dims) returns a modified copy of type_obj = coder.typeof(v) with upper bound size specified by sz and variable dimensions specified by variable_dims.

type_obj = coder.typeof(v,'Gpu', true) creates an object that is derived from coder.Type to represent v as a GPU input type for code generation. This option requires a valid GPU Coder license.

type_obj = coder.typeof(type_obj) returns type_obj itself.

Examples

Create Type for a Matrix

Create a type for a simple fixed-size 5x6 matrix of doubles.

```
coder.typeof(ones(5,6))
```

```
ans = 

coder.PrimitiveType
   5×6 double
```

```
coder.typeof(0,[5 6])
```

```
ans = 
```

```
coder.PrimitiveType
   5×6 double
```

Create a type for a variable-size matrix of doubles.

```
coder.typeof(ones(3,3),[],1)

ans =

coder.PrimitiveType
   :3×:3 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix with fixed-size and variable-size dimensions.

```
coder.typeof(0,[2,3,4],[1 0 1])

ans =

coder.PrimitiveType
   :2×3×:4 double
```

```
coder.typeof(10,[1 5],1)

ans =

coder.PrimitiveType
   1×:5 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with fixed size.

```
coder.typeof(10,[inf,3])

ans =

coder.PrimitiveType
   :inf×3 double
% ':' indicates variable-size dimensions
```

Create a type for a matrix of doubles, first dimension unbounded, second dimension with variable size that has an upper bound of 3.

```
coder.typeof(10,[inf,3],[0 1])

ans =

coder.PrimitiveType
   :inf×:3 double
```

Convert a fixed-size matrix to a variable-size matrix.

```
coder.typeof(ones(5,5),[],1)

 ans =

coder.PrimitiveType
```

```
   :5×:5 double
% ':' indicates variable-size dimensions
```

Create Type for a Structure

Create a type for a structure with a variable-size field.

```
x.a = coder.typeof(0,[3 5],1);
x.b = magic(3);
coder.typeof(x)
```

```
ans = 

coder.StructType
   1×1 struct
      a: :3×:5 double
      b: 3×3 double
% ':' indicates variable-size dimensions
```

Create a nested structure (a structure as a field of another structure).

```
S = struct('a',double(0),'b',single(0));
SuperS.x = coder.typeof(S);
SuperS.y = single(0);
coder.typeof(SuperS)
```

```
ans = 

coder.StructType
   1×1 struct
      x: 1×1 struct
         a: 1×1 double
         b: 1×1 single
      y: 1×1 single
```

Create a structure containing a variable-size array of structures as a field.

```
S = struct('a',double(0),'b',single(0));
SuperS.x = coder.typeof(S,[1 inf],[0 1]);
SuperS.y = single(0);
coder.typeof(SuperS)
```

```
ans = 

coder.StructType
   1×1 struct
      x: 1×:inf struct
         a: 1×1 double
         b: 1×1 single
      y: 1×1 single
% ':' indicates variable-size dimensions
```

Create Type for a Cell Array

Create a type for a homogeneous cell array with a variable-size field.

```
a = coder.typeof(0,[3 5],1);
b = magic(3);
coder.typeof({a b})
```

```
ans = 

coder.CellType
   1×2 homogeneous cell 
      base: :3×:5 double
% ':' indicates variable-size dimensions
```

Create a type for a heterogeneous cell array.

```
a = coder.typeof('a');
b = coder.typeof(1);
coder.typeof({a b})
```

```
ans = 

coder.CellType
   1×2 heterogeneous cell 
      f1: 1×1 char
      f2: 1×1 double
```

Create a variable-size homogeneous cell array type from a cell array that has the same class but different sizes.

1. Create a type for a cell array that contains two character vectors with different sizes. The cell array type is heterogeneous.

```
coder.typeof({'aa','bbb'})
```

```
ans = 

coder.CellType
   1×2 heterogeneous cell 
      f1: 1×2 char
      f2: 1×3 char
```

2. Create a type by using the same cell array input. This time, specify that the cell array type has variable-size dimensions. The cell array type is homogeneous.

```
coder.typeof({'aa','bbb'},[1,10],[0,1])
```

```
ans = 

coder.CellType
   1×:10 locked homogeneous cell 
      base: 1×:3 char
% ':' indicates variable-size dimensions
```

Create Type for a Value Class Object

Change a fixed-size array to a bounded, variable-size array.

Create a type for a value class object.

1. Create this value class:

```
classdef mySquare
    properties
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end
```

2. Create an object of `mySquare`.

```
sq_obj = coder.typeof(mySquare(4))
```

```
sq_obj =

coder.ClassType
   1×1 mySquare
      side: 1×1 double
```

3. Create a type for an object that has the same properties as `sq_obj`.

```
t = coder.typeof(sq_obj)
```

```
t =

coder.ClassType
   1×1 mySquare
      side: 1×1 double
```

Alternatively, you can create the type from the class definition:

```
t = coder.typeof(mySquare(4))
```

```
t =

coder.ClassType
   1×1 mySquare
      side: 1×1 double
```

Create Type for a String Scalar

Define a string scalar. For example:

```
s = "mystring";
```

Create a type from `s`.

```
t = coder.typeof(s);
```

Assign the `StringLength` property of the type object the upper bound of the string length and set `VariableStringLength` to `true`. Specify that type object `t` is variable-size with an upper bound of 10.

```
t.StringLength = 10;
t.VariableStringLength = true;
```

To specify that `t` is variable-size without an upper bound:

```
t.StringLength = Inf;
```

This automatically sets the `VariableStringLength` property to `true`.

Pass the type to `codegen` by using the `-args` option.

```
codegen myFunction -args {t}
```

Input Arguments

v — Set of values representing input parameter types
numeric array | character vector | string | struct | cell array

`v` can be a MATLAB numeric, logical, char, enumeration, or fixed-point array. `v` can also be a cell array, structure, or value class that contains the previous types.

When `v` is a cell array whose elements have the same classes but different sizes, if you specify variable-size dimensions, `coder.typeof` creates a homogeneous cell array type. If the elements have different classes, `coder.typeof` reports an error.

Example: `coder.typeof(ones(5,6));`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

sz — Dimension of type object
row vector of integer values

Size vector specifying each dimension of type object.

If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds of `v` do not change.

If size is not specified, `sz` takes the default dimension of `v`.

Example: `coder.typeof(0,[5,6]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

variable_dims — Variable or fixed dimension
row vector of logical values

Logical vector that specifies whether each dimension is variable size (true) or fixed size (false). For a cell array, if the elements have different classes, you cannot specify variable-size dimensions.

If you do not specify the `variable_dims` input parameter, the bounded dimensions of the type are fixed.

A scalar `variable_dims` applies to all dimensions. However, if `variable_dims` is `1`, the size of a singleton dimension remains fixed.

Example: `coder.typeof(0,[2,3,4],[1 0 1]);`

Data Types: `logical`

`type_obj` — Type object
coder.Type object

`coder.Type` object to represent the type of `v` for code generation.

Example: `type_obj = coder.typeof(ones(5,6));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

Output Arguments

`type_obj` — Type object
`coder.Type` object

`coder.Type` object to represent the type of `v` for code generation.

Example: `type_obj = coder.typeof(ones(5,6));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`
Complex Number Support: Yes

Limitations

- For sparse matrices, `coder.typeof` drops upper bounds for variable-size dimensions.
- For representing GPU arrays, only bounded numeric and logical base types are supported. Scalar GPU arrays, structures, cell-arrays, classes, enumerated types, character, half-precision and fixed-point data types are not supported.
- When using `coder.typeof` to represent GPU arrays, the memory allocation (malloc) mode property of the GPU code configuration object must be set to be `'discrete'`.

Tips

- `coder.typeof` fixes the size of a singleton dimension unless the `variable_dims` argument explicitly specifies that the singleton dimension has a variable size.

  For example, the following code specifies a 1-by-:10 double. The first dimension (the singleton dimension) has a fixed size. The second dimension has a variable size.

  ```
  t = coder.typeof(5,[1 10],1)
  ```

  By contrast, this code specifies a :1-by-:10 double. Both dimensions have a variable size.

  ```
  t = coder.typeof(5,[1 10],[1 1])
  ```

Note For a MATLAB Function block, singleton dimensions of input or output signals cannot have a variable size.

- If you are already specifying the type of an input variable by using a type function, do not use `coder.typeof` unless you also want to specify the size. For instance, instead of `coder.typeof(single(0))`, use the syntax `single(0)`.
- For cell array types, `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous.

  If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type.

  If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type.

  For some cell arrays, classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type where the first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the coder.CellType `makeHomogeneous` or `makeHeterogeneous` methods to make a type with the classification that you want. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

  The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as heterogeneous and homogeneous. You cannot later use one of these methods to create a copy that has a different classification.
- During code generation with GPU array types, if one input to the entry-point function is of the GPU array type, then the output variables are all GPU array types, provided they are supported for GPU code generation. For example. if the entry-point function returns a `struct` and because `struct` is not supported, the generated code returns a CPU output. However, if a supported matrix type is returned, then the generated code returns a GPU output.

Version History

Introduced in R2011a

See Also

`coder.newtype` | `coder.resize` | `coder.Type` | `coder.ArrayType` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.CellType` | `fiaccel` | `coder.OutputType`

Topics

“Define Input Properties by Example at the Command Line”
“Specify Cell Array Inputs at the Command Line”
“Specify Objects as Inputs”
“Define String Scalar Inputs”
“Create and Edit Input Types by Using the Coder Type Editor”

coder.unroll

Unroll for-loop by making a copy of the loop body for each loop iteration

Syntax

```
coder.unroll()
coder.unroll(flag)
```

Description

`coder.unroll()` unrolls a `for`-loop. The `coder.unroll` call must be on a line by itself immediately preceding the `for`-loop that it unrolls.

Instead of producing a `for`-loop in the generated code, loop unrolling produces a copy of the `for`-loop body for each loop iteration. In each iteration, the loop index becomes constant. To unroll a loop, the code generator must be able to determine the bounds of the `for`-loop.

For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can increase code generation time significantly and generate inefficient code.

`coder.unroll` is ignored outside of code generation.

`coder.unroll(flag)` unrolls a `for`-loop if `flag` is `true`. `flag` is evaluated at code generation time. The `coder.unroll` call must be on a line by itself immediately preceding the `for`-loop that it unrolls.

Examples

Unroll a for-loop

To produce copies of a `for`-loop body in the generated code, use `coder.unroll`.

In one file, write the entry-point function `call_getrand` and a local function `getrand`. `getrand` unrolls a `for`-loop that assigns random numbers to an n-by-1 array. `call_getrand` calls `getrand` with the value 3.

```
function z = call_getrand
%#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
coder.unroll();
for i = 1:n
    y(i) = rand();
end
end
```

Generate a static library.

```
codegen -config:lib call_getrand -report
```

In the generated code, the code generator produces a copy of the `for`-loop body for each of the three loop iterations.

```
static void getrand(double y[3])
{
  y[0] = b_rand();
  y[1] = b_rand();
  y[2] = b_rand();
}
```

Control `for`-loop Unrolling with Flag

Control loop unrolling by using `coder.unroll` with the `flag` argument.

In one file, write the entry-point function `call_getrand_unrollflag` and a local function `getrand_unrollflag`. When the number of loop iterations is less than 10, `getrand_unrollflag` unrolls the `for`-loop. `call_getrand` calls `getrand` with the value 50.

```
function z = call_getrand_unrollflag
%#codegen
z = getrand_unrollflag(50);
end

function y = getrand_unrollflag(n)
coder.inline('never');
unrollflag = n < 10;
y = zeros(n, 1);
coder.unroll(unrollflag)
for i = 1:n
    y(i) = rand();
end
end
```

Generate a static library.

```
codegen -config:lib call_getrand_unrollflag -report
```

```
static void getrand_unrollflag(double y[50])
{
  int i;
  for (i = 0; i < 50; i++) {
    y[i] = b_rand();
  }
}
```

The number of iterations is not less than 10. Therefore, the code generator does not unroll the `for`-loop. It produces a `for`-loop in the generated code.

Use Legacy Syntax to Unroll for-Loop

- ```
  function z = call_getrand
  %#codegen
  z = getrand(3);
  end

  function y = getrand(n)
  coder.inline('never');
  y = zeros(n, 1);
  for i = coder.unroll(1:n)
      y(i) = rand();
  end
  end
  ```

Use Legacy Syntax to Control for-Loop Unrolling

- ```
  function z = call_getrand_unrollflag
  %#codegen
  z = getrand_unrollflag(50);
  end

  function y = getrand_unrollflag(n)
  coder.inline('never');
  unrollflag = n < 10;
  y = zeros(n, 1);
  for i = coder.unroll(1:n, unrollflag)
      y(i) = rand();
  end
  end
  ```

Input Arguments

`flag` — Indicates whether to unroll the `for`-loop

true (default) | false

When `flag` is `true`, the code generator unrolls the `for`-loop. When `flag` is `false`, the code generator produces a `for`-loop in the generated code. `flag` is evaluated at code generation time.

Tips

- Sometimes, the code generator unrolls a `for`-loop even though you do not use `coder.unroll`. For example, if a `for`-loop indexes into a heterogeneous cell array or into `varargin` or `varargout`, the code generator unrolls the loop. By unrolling the loop, the code generator can determine the value of the index for each loop iteration. The code generator uses heuristics to determine when to unroll a `for`-loop. If the heuristics fail to identify that unrolling is warranted, or if the number of loop iterations exceeds a limit, code generation fails. In these cases, you can force loop unrolling by using `coder.unroll`. See "Nonconstant Index into varargin or varargout in a for-Loop".

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`coder.inline`

Topics
"Nonconstant Index into varargin or varargout in a for-Loop"

coder.varsize

Package: `coder`

Declare variable-size data

Syntax

```
coder.varsize(varName1,...,varNameN)
coder.varsize(varName1,...,varNameN,ubounds)
coder.varsize(varName1,...,varNameN,ubounds,dims)
```

Description

`coder.varsize(varName1,...,varNameN)` declares that the variables named `varName1,...,varNameN` have a variable size. The declaration instructs the code generator to allow the variables to change size during execution of the generated code. With this syntax, you do not specify the upper bounds of the dimensions of the variables or which dimensions can change size. The code generator computes the upper bounds. All dimensions, except singleton dimensions on page 4-233, are allowed to change size.

Use `coder.varsize` according to these restrictions and guidelines:

- Use `coder.varsize` inside a MATLAB function intended for code generation.
- The `coder.varsize` declaration must precede the first use of a variable. For example:

  ```
  ...
  x = 1;
  coder.varsize('x');
  disp(size(x));
  ...
  ```
- Use `coder.varsize` to declare that an output argument has a variable size or to address size mismatch errors. Otherwise, to define variable-size data, use the methods described in "Define Variable-Size Data for Code Generation".

Note For MATLAB Function blocks, to declare variable-size output variables, use the **Symbols** pane and Property Inspector. See "Declare Variable-Size MATLAB Function Block Variables". If you provide upper bounds in a `coder.varsize` declaration, the upper bounds must match the upper bounds in the Property Inspector.

For more restrictions and guidelines, see "Limitations" on page 4-232 and "Tips" on page 4-233.

`coder.varsize(varName1,...,varNameN,ubounds)` also specifies an upper bound for each dimension of the variables. All variables must have the same number of dimensions. All dimensions, except singleton dimensions on page 4-233, are allowed to change size.

`coder.varsize(varName1,...,varNameN,ubounds,dims)` also specifies an upper bound for each dimension of the variables and whether each dimension has a fixed size or a variable size. If a dimension has a fixed size, then the corresponding `ubound` element specifies the fixed size of the dimension. All variables have the same fixed-size dimensions and the same variable-size dimensions.

The code generator uses a colon prefix to denote a variable-size dimension. For example, if the size of an array `A` is denoted as `3x:5x:Inf`, then:

- The first dimension has a fixed size `3`
- The second dimension is variable-size with an upper bound `5`
- The third dimension is variable-size and unbounded

Examples

Address Size Mismatch Error by Using `coder.varsize`

After a variable is used (read), changing the size of the variable can cause a size mismatch error. Use `coder.varsize` to specify that the size of the variable can change.

Code generation for the following function produces a size mismatch error because `x = 1:10` changes the size of the second dimension of `x` after the line `y = size(x)` that uses `x`.

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
y = size(x);
if n > 10
    x = 1:10;
end
```

To declare that `x` can change size, use `coder.varsize`.

```
function [x,y] = usevarsize(n)
%#codegen
x = 1;
coder.varsize('x');
y = size(x);
if n > 10
    x = 1:10;
end
```

If you remove the line `y = size(x)`, you no longer need the `coder.varsize` declaration because `x` is not used before its size changes.

Declare Variable-Size Array with Upper Bounds

Specify that `A` is a row vector of size `1x:20`. This denotes that the second dimension of `A` has a variable size with an upper bound of 20.

```
function fcn()
...
coder.varsize('A',[1 20]);
...
end
```

When you do not provide `dims`, all dimensions, except singleton dimensions, have a variable size.

Declare Variable-Size Array with a Mix of Fixed and Variable Dimensions

Specify that A is an array with size `3x:20`. This denotes that the first dimension has a fixed size of three and whose second dimension has a variable size with an upper bound of 20.

```
function fcn()
...
coder.varsize('A',[3 20], [0 1] );
...
end
```

Declare Variable-Size Structure Fields

In this function, the statement `coder.varsize('data.values')` declares that the field `values` inside each element of `data` has a variable size.

```
function y = varsize_field()
%#codegen

d = struct('values', zeros(1,0), 'color', 0);
data = repmat(d, [3 3]);
coder.varsize('data.values');

for i = 1:numel(data)
    data(i).color = rand-0.5;
    data(i).values = 1:i;
end

y = 0;
for i = 1:numel(data)
    if data(i).color > 0
        y = y + sum(data(i).values);
    end
end
```

Declare Variable-Size Cell Array

Specify that cell array `C` has a fixed-size first dimension and variable-size second dimension with an upper bound of three. The `coder.varsize` declaration must precede the first use of `C`.

```
...
C = {1 [1 2]};
coder.varsize('C', [1 3], [0 1]);
y = C{1};
...
end
```

Without the `coder.varsize` declaration, `C` is a heterogeneous cell array whose elements have the same class and different sizes. With the `coder.varsize` declaration, `C` is a homogeneous cell array whose elements have the same class and maximum size.

- The cell array C is of size `1x:3`. The colon denotes that the second dimension of C is variable-size with an upper bound of 3.

- Each element of C is a `1x:2` array.

Declare That a Cell Array Has Variable-Size Elements

Specify that the elements of cell array `C` are `1x:5` vectors. This denotes that the elements each have a fixed-size first dimension and variable-size second dimension with an upper bound of 5.

```
...
C = {1 2 3};
coder.varsize('C{:}', [1 5], [0 1]);
C = {1, 1:5, 2:3};
...
```

You can also specify a specific element of a cell array to be variable-size. For example, in a 1-by-3 cell array `x`, declare the first element `x{1}` to be a `1x:10` row vector.

```
...
x = {1,2,3};
coder.varsize('x{1}', [1 10]);
...
```

Input Arguments

`varName1,...,varNameN` — Names of variables to declare as having a variable size
character vectors | string scalars

Names of variables to declare as having a variable size, specified as one or more character vectors or string scalars.

Example: `coder.varsize('x','y')`

`ubounds` — Upper bounds for array dimensions
[] (default) | vector of integer constants

Upper bounds for array dimensions, specified as a vector of integer constants.

When you do not specify `ubounds`, the code generator computes the upper bound for each variable. If the `ubounds` element corresponds to a fixed-size dimension, the value is the fixed size of the dimension.

Example: `coder.varsize('x','y',[1 2])`

`dims` — Indication of whether each dimension has a fixed size or a variable size
logical vector

Indication of whether each dimension has a fixed size or a variable size, specified as a logical vector. Dimensions that correspond to 0 or `false` in `dims` have a fixed size. Dimensions that correspond to 1 or `true` have a variable size.

When you do not specify `dims`, the dimensions have a variable size, except for the singleton dimensions.

Example: `coder.varsize('x','y',[1 2], [0 1])`

Limitations

- The `coder.varsize` declaration instructs the code generator to allow the size of a variable to change. It does not change the size of the variable. Consider this code:

  ```
  ...
  x = 7;
  coder.varsize('x', [1,5]);
  disp(size(x));
  ...
  ```

  After the `coder.varsize` declaration, `x` is still a 1-by-1 array. You cannot assign a value to an element beyond the current size of `x`. For example, this code produces a run-time error because the index 3 exceeds the dimensions of `x`.

  ```
  ...
  x = 7;
  coder.varsize('x', [1,5]);
  x(3) = 1;
  ...
  ```
- `coder.varsize` is not supported for a function input argument. Instead:
  - If the function is an entry-point function, specify that an input argument has a variable size by using `coder.typeof` at the command line. Alternatively, specify that an entry-point function input argument has a variable size by using the **Define Input Types** step of the app.
  - If the function is not an entry-point function, use `coder.varsize` in the calling function with the variable that is the input to the called function.
- For sparse matrices, `coder.varsize` drops upper bounds for variable-size dimensions.
- Limitations for using `coder.varsize` with cell arrays:
  - A cell array can have a variable size only if it is homogeneous. When you use `coder.varsize` with a heterogeneous cell array, the code generator tries to make the cell array homogeneous. The code generator tries to find a class and maximum size that apply to all elements of the cell array. For example, consider the cell array `c = {1, [2 3]}`. Both elements can be represented by a double type whose first dimension has a fixed size of 1 and whose second dimension has a variable size with an upper bound of 2. If the code generator cannot find a common class and a maximum size, code generation fails. For example, consider the cell array `c = {'a',[2 3]}`. The code generator cannot find a class that can represent both elements because the first element is `char` and the second element is `double`.
  - If you use the `cell` function to define a fixed-size cell array, you cannot use `coder.varsize` to specify that the cell array has a variable size. For example, this code causes a code generation error because `x = cell(1,3)` makes `x` a fixed-size,1-by-3 cell array.

    ```
    ...
    x = cell(1,3);
    coder.varsize('x',[1 5])
    ...
    ```

    You can use `coder.varsize` with a cell array that you define by using curly braces. For example:

    ```
    ...
    x = {1 2 3};
    coder.varsize('x',[1 5])
    ...
    ```

- To create a variable-size cell array by using the `cell` function, use this code pattern:

  ```
  function mycell(n)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  end
  end
  ```

  See “Definition of Variable-Size Cell Array by Using cell”.

  To specify upper bounds for the cell array, use `coder.varsize`.

  ```
  function mycell(n)
  %#codegen
  x = cell(1,n);
  for i = 1:n
      x{i} = i;
  coder.varsize('x',[1,20]);
  end
  end
  ```
- `coder.varsize` is not supported for:
  - Global variables
  - MATLAB classes or class properties
  - String scalars

More About

Singleton Dimension

Dimension for which `size(A,dim) = 1`.

Tips

- In a code generation report or a MATLAB Function report, a colon (:) indicates that a dimension has a variable size. For example, a size of `1x:2` indicates that the first dimension has a fixed size of one and the second dimension has a variable size with an upper bound of two.
- If you use `coder.varsize` to specify that the upper bound of a dimension is 1, by default, the dimension has a fixed size of 1. To specify that the dimension can be 0 (empty array) or 1, set the corresponding element of the `dims` argument to `true`. For example, this code specifies that the first dimension of `x` has a fixed size of 1 and the other dimensions have a variable size of 5.

  ```
  coder.varsize('x',[1,5,5])
  ```

  In contrast, this code specifies that the first dimension of `x` has an upper bound of 1 and has a variable size (can be 0 or 1).

  ```
  coder.varsize('x',[1,5,5],[1,1,1])
  ```
- If you use input variables or the result of a computation using input variables to specify the size of an array, it is declared as variable-size in the generated code. Do not re-use `coder.varsize` on the array, unless you also want to specify an upper bound for its size.

- If you do not specify upper bounds with a `coder.varsize` declaration and the code generator is unable to determine the upper bounds, the generated code uses dynamic memory allocation. Dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, specify the upper bounds by providing the `ubounds` argument.

Version History

Introduced in R2011a

See Also

`coder.typeof`

Topics

"Code Generation for Variable-Size Arrays"
"Incompatibilities with MATLAB in Variable-Size Support for Code Generation"

colon, :

Create vectors, array subscripting

Syntax

```
y = j:k
y = j:i:k
```

Description

`y = j:k` returns a regularly-spaced vector, [j, j+1 ,..., k]. `j:k` is empty when `j > k`.

At least one of the colon operands must be a `fi` object. All colon operands must have integer values. All the fixed-point operands must be binary-point scaled. Slope-bias scaling is not supported. If any of the operands is complex, the `colon` function generates a warning and uses only the real part of the operands.

`y = colon(j,k)` is the same as `y = j:k`.

`y = j:i:k` returns a regularly-spaced vector, `[j,j+i,j+2i, ...,j+m*i]`, where `m = fix((k-j)/i)`. `y = j:i:k` returns an empty matrix when`i == 0`, `i > 0` and `j > k`, or `i < 0` and `j < k`.

Examples

Use fi as a Colon Operator

When you use `fi` as a colon operator, all colon operands must have integer values.

```
a = fi(1,0,3,0);
b = fi(2,0,8,0);
c = fi(12,0,8,0);
x = a:b:c
```

```
x =
     1     3     5     7     9    11

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

Because all the input operands are unsigned, `x` is unsigned and the word length is 8. The fraction length of the resulting vector is always 0.

Use the colon Operator With Signed and Unsigned Operands

```
a= fi(int8(-1));
b = uint8(255);
```

```
c = a:b;
len = c.WordLength
```

```
len = 9
```

```
signedness = c.Signedness
```

```
signedness =
'Signed'
```

The word length of `c` requires an additional bit to handle the intersection of the ranges of `int8` and `uint8`. The data type of `c` is signed because the operand `a` is signed.

Create a Vector of Decreasing Values

If the beginning and ending operands are unsigned, the increment operand can be negative.

```
x = fi(4,false):-1:1
```

```
x =
     4     3     2     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 0
```

Use the colon Operator With Floating-Point and fi Operands

If any of the operands is floating-point, the output has the same word length and signedness as the `fi` operand

```
x = fi(1):10
```

```
x =
     1     2     3     4     5     6     7     8     9    10

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

`x = fi(1):10` is equivalent to `fi(1:10, true, 16, 0)` so `x` is signed and its word length is 16 bits.

Rewrite Code That Uses Non-Integer Operands

If your code uses non-integer operands, rewrite the colon expression so that the operands are integers.

The following code does not work because the colon operands are not integer values.

```
Fs = fi(100);
n = 1000;
t  = (0:1/Fs:(n/Fs - 1/Fs));
```

Rewrite the colon expression to use integer operands.

```
Fs = fi(100);
n = 1000;
t  = (0:(n-1))/Fs;
```

All Colon Operands Must Be in the Range of the Data Type

If the value of any of the colon operands is outside the range of the data type used in the colon expression, MATLAB generates an error.

```
y = fi(1,true,8,0):256
```

MATLAB generates an error because 256 is outside the range of `fi(1,true, 8,0)`. This behavior matches the behavior for built-in integers. For example, `y = int8(1):256` generates the same error.

Input Arguments

j — Beginning operand
real scalar

Beginning operand, specified as a real scalar integer-valued `fi` object or built-in numeric type.

If you specify non-scalar arrays, MATLAB interprets `j:i:k` as `j(1):i(1):k(1)`.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

i — Increment
1 (default) | real scalar

Increment, specified as a real scalar integer-valued `fi` object or built-in numeric type. Even if the beginning and end operands, `j` and `k`, are both unsigned, the increment operand `i` can be negative.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

k — Ending operand
real scalar

Ending operand, specified as a real scalar integer-valued `fi` object or built-in numeric type.

Data Types: `fi` |`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

y — Regularly-spaced vector
real vector

Fixed-Point Designer determines the data type of the `y` using the following rules:

- The data type covers the union of the ranges of the fixed-point types of the input operands.
- If either the beginning or ending operand is signed, the resulting data type is signed. Otherwise, the resulting data type is unsigned.
- The word length of `y` is the smallest value such that the fraction length is 0 and the real-world value of the least-significant bit is 1.
- If any of the operands is floating-point, the word length and signedness of `y` is derived from the `fi` operand.
- If any of the operands is a scaled double, `y` is a scaled double.
- The `fimath` of `y` is the same as the `fimath` of the input operands.
- If all the `fi` objects are of data type `double`, the data type of `y` is `double`. If all the `fi` objects are of data type `single`, the data type of `y` is `single`. If there are both `double` and `single` inputs, and no fixed-point inputs, the output data type is `single`.

Version History

Introduced in R2013b

See Also

`colon` | `fi`

complex

Construct complex `fi` object from real and imaginary parts

Syntax

```
c = complex(a,b)
c = complex(x)
```

Description

`c = complex(a,b)` creates a complex output, `c`, from two real inputs, such that `c = a + bi`.

When `b` is all zero, `c` is complex with an all-zero imaginary part. This is in contrast to the addition of `a + 0i`, which returns a strictly real result.

`c = complex(x)` returns the complex equivalent of `x`, such that `isreal(c)` returns logical `0` (`false`).

- If `x` is real, then `c` is `x + 0i`.
- If `x` is complex, then `c` is identical to `x`.

Examples

Complex Scalar from Two Real Scalars

Use the `complex` function to create the complex scalar, `3 + 4i`.

```
a = fi(3,1,16,12);
b = fi(4,0,8);
c = complex(a,b)
```

```
c = 

   3.0000 + 4.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

The output, `c`, has the same `numerictype` and `fimath` properties as the input `fi` object, `a`.

Complex Vector from One Real Vector

Create a complex `fi` vector with a zero imaginary part.

```
x = fi([1;2;3;4]);
c = complex(x)
```

```
c = 

   1.0000 + 0.0000i
   2.0000 + 0.0000i
   3.0000 + 0.0000i
   4.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

Verify that `c` is complex.

```
isreal(c)
```

```
ans = 

  logical

   0
```

Input Arguments

a — Real component
scalar | vector | matrix | multidimensional array

Real component, specified as a `fi` scalar, vector, matrix, or multidimensional array.

The size of `a` must match the size of `b`, unless one is a scalar. If either `a` or `b` is a scalar, MATLAB expands the scalar to match the size of the other input.

Data Types: `fi`

b — Imaginary component
scalar | vector | matrix | multidimensional array

Imaginary component, specified as a `fi` scalar, vector, matrix, or multidimensional array.

The size of `b` must match the size of `a`, unless one is a scalar. If either `a` or `b` is a scalar, MATLAB expands the scalar to match the size of the other input.

Data Types: `fi`

x — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a `fi` scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

Output Arguments

c — Complex array
scalar | vector | matrix | multidimensional array

Complex array, returned as a `fi` scalar, vector, matrix, or multidimensional array.

The size of `c` is the same as the input arguments.

The output `fi` object, `c`, has the same `numerictype` and `fimath` properties as the input `fi` object, `a`.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`fi` | `fimath` | `numerictype`

conj

Complex conjugate of `fi` object

Syntax

`conj(a)`

Description

`conj(a)` is the complex conjugate of `fi` object `a`.

When `a` is complex,

$$\mathrm{conj}(a) = \mathrm{real}(a) - i \times \mathrm{imag}(a)$$

The `numerictype` and `fimath` properties associated with the input `a` are applied to the output.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`complex`

conv

Convolution and polynomial multiplication of `fi` objects

Syntax

```
c = conv(a,b)
c = conv(a,b,shape)
```

Description

`c = conv(a,b)` returns the convolution of input vectors `a` and `b`, at least one of which must be a `fi` object.

`c = conv(a,b,shape)` returns a subsection of the convolution, as specified by `shape`.

Examples

Convolution of 22-Sample Sequence with 16-Tap FIR Filter

Find the convolution of a 22-sample sequence with a 16-tap FIR filter.

`x` is a 22-sample sequence of signed values with a word length of 16 bits and a fraction length of 15 bits. `h` is the 16-tap FIR filter.

```
u = (pi/4)*[1 1 1 -1 -1 -1 1 -1 -1 1 -1];
x = fi(kron(u,[1 1]));
h = firls(15, [0 .1 .2 .5]*2, [1 1 0 0]);
```

Because `x` is a `fi` object, you do not need to cast `h` into a `fi` object before performing the convolution operation. The `conv` function does this automatically using best-precision scaling.

Use the `conv` function to convolve the two vectors.

```
y = conv(x,h);
```

The operation results in a signed `fi` object `y` with a word length of 36 bits and a fraction length of 31 bits. The default `fimath` properties associated with the inputs determine the `numerictype` of the output. The output does not have a local `fimath`.

Central Part of Convolution of Two `fi` Vectors

Create two `fi` vectors. Find the central part of the convolution of `a` and `b` that is the same size as `a`.

```
a = fi([-1 2 3 -2 0 1 2]);
b = fi([2 4 -1 1]);
c = conv(a,b,'same')
```

```
c =
```

```
   15     5    -9     7     6     7    -1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 34
        FractionLength: 25
```

`c` has a length of 7. The full convolution would be of length `length(a)+length(b)-1`, which in this example would be 10.

Input Arguments

`a,b` — Input vectors
vectors

Input vectors, specified as either row or column vectors.

If either input is a built-in data type, `conv` casts it into a `fi` object using best-precision rules before the performing the convolution operation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`shape` — Subset of convolution
`'full'` (default) | `'same'` | `'valid'`

Subset of convolution, specified as one of these values:

- `'full'` — Returns the full convolution. This option is the default shape.
- `'same'` — Returns the central part of the convolution that is the same size as input vector `a`.
- `'valid'` — Returns only those parts of the convolution that the function computes without zero-padded edges. Using this option, the length of output vector `c` is `max(length(a)-max(0,length(b)-1),0)`.

Data Types: `char`

More About

Convolution

The convolution of two vectors, `u` and `v`, represents the area of overlap under the points as `v` slides across `u`. Algebraically, convolution is the same operation as multiplying polynomials whose coefficients are the elements of `u` and `v`.

Let `m = length(u)` and `n = length(v)` . Then `w` is the vector of length `m+n-1` whose `k`th element is

The sum is over all the values of `j` that lead to legal subscripts for `u(j)` and `v(k-j+1)`, specifically `j` = `max(1,k+1-n):1:min(k,m)`. When `m` = `n`, this gives

```
w(1) = u(1)*v(1)
w(2) = u(1)*v(2)+u(2)*v(1)
```

```
w(3) = u(1)*v(3)+u(2)*v(2)+u(3)*v(1)
...
w(n) = u(1)*v(n)+u(2)*v(n-1)+ ... +u(n)*v(1)
...
w(2*n-1) = u(n)*v(n)
```

Algorithms

The `fimath` properties associated with the inputs determine the `numerictype` properties of output `fi` object `c`:

- If either `a` or `b` has a local `fimath` object, `conv` uses that `fimath` object to compute intermediate quantities and determine the `numerictype` properties of `c`.
- If neither `a` nor `b` have an attached `fimath`, `conv` uses the default `fimath` to compute intermediate quantities and determine the `numerictype` properties of `c`.

If either input is a built-in data type, `conv` casts it into a `fi` object using best-precision rules before the performing the convolution operation.

The output `fi` object `c` always uses the default `fimath`.

Version History

Introduced in R2009b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `SpecifyPrecision` or `KeepLSB`.
- For variable-sized signals, you might see different results between generated code and MATLAB.
  - In the generated code, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath`.
  - In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the `ProductMode` of the governing `fimath`.

See Also

`conv`

convergent

Round toward nearest integer with ties rounding to nearest even integer

Syntax

```
y = convergent(a)
y = convergent(x)
```

Description

`y = convergent(a)` rounds `fi` object `a` to the nearest integer. In the case of a tie, `convergent(a)` rounds to the nearest even integer.

`y = convergent(x)` rounds the elements of `x` to the nearest integer. In the case of a tie, `convergent(x)` rounds to the nearest even integer.

Examples

Use Convergent Rounding on Signed `fi` Object

The following example demonstrates how the `convergent` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi,1,8,3)
```

```
a = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

```
y = convergent(a)
```

```
y = 
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

The following example demonstrates how the `convergent` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a = 
    0.0249
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
y = convergent(a)
```

```
y = 
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Compare Rounding Methods

The functions `convergent`, `nearest`, and `round` differ in the way they treat values whose least significant digit is 5.

- The `convergent` function rounds ties to the nearest even integer.
- The `nearest` function rounds ties to the nearest integer toward positive infinity.
- The `round` function rounds ties to the nearest integer with greater absolute value.

This example illustrates these differences for a given input, a.

```
a = fi([-3.5:3.5]');
y = [a convergent(a) nearest(a) round(a)]
```

```
y = 
   -3.5000   -4.0000   -3.0000   -4.0000
   -2.5000   -2.0000   -2.0000   -3.0000
   -1.5000   -2.0000   -1.0000   -2.0000
   -0.5000         0         0   -1.0000
    0.5000         0    1.0000    1.0000
    1.5000    2.0000    2.0000    2.0000
    2.5000    2.0000    3.0000    3.0000
    3.5000    3.9999    3.9999    3.9999

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

a — Input `fi` array
scalar | vector | matrix | multidimensional array

Input `fi` array, specified as scalar, vector, matrix, or multidimensional array.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`convergent` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Data Types: `fi`
Complex Number Support: Yes

x — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

For complex inputs, the real and imaginary parts are rounded independently.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`
Complex Number Support: Yes

Algorithms

- `y` and `a` have the same `fimath` object and `DataType` property.
- When the `DataType` property of `a` is `single`, or `double`, the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

Version History
Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`ceil` | `fix` | `floor` | `nearest` | `round`

Topics
"Precision and Range"

convertToSingle

Convert double-precision MATLAB code to single-precision MATLAB code

Syntax

```
convertTosingle options fcn_1, ..., fcn_n
convertTosingle options fcn_1, -args args_1 ,..., fcn_n -args args_n
```

Description

`convertTosingle options fcn_1, ..., fcn_n` generates single-precision MATLAB code from the specified function or functions. When you use this syntax, you must provide a test file that `convertToSingle` can use to determine the properties of the input parameters. To specify the test file, use `coder.config('single')` to create a `coder.SingleConfig` object. Specify the `TestBenchName` property.

`convertTosingle options fcn_1, -args args_1 ,..., fcn_n -args args_n` specifies the properties of the input arguments.

Examples

Convert to Single Precision and Validate Using a Test File

Generate single-precision code from a double-precision function `myfun.m`. Specify a test file for determining the argument properties and for verification of the converted types. Plot the error between the double-precision and single-precision values.

```
scfg = coder.config('single');
scfg.TestBenchName = 'myfun_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
convertToSingle -config scfg myfun
```

Convert Multiple Functions to Single Precision with the Default Configuration

Convert `myfun1.m` and `myfun2.m` to single precision. Specify that `myfun1` has a double scalar argument and `myfun2` has a 2x3 double argument.

```
convertToSingle -config scfg myfun1 -args {0} myfun2 -args {zeros(2, 3)}
```

Specify Input Argument Properties

Generate single-precision code from a double-precision function, `myfun.m`, whose first argument is double scalar and whose second argument is 2x3 double.

```
convertToSingle  myfun -args {0, zeros(2, 3)}
```

Input Arguments

`fcn` — Function name

character vector

MATLAB function from which to generate single-precision code.

`args` — Argument properties

cell array of types or example values.

Definition of the size, class, and complexity of the input arguments specified as a cell array of types or example values. To create a type, use `coder.typeof`.

`options` — options for single-precision conversion

`-config` | `-globals`

Specify one of the following single-conversion options.

`-config` *`config_object`*	Specify the configuration object to use for conversion of double-precision MATLAB code to single-precision MATLAB code. To create the configuration object, use `coder.config('single');` If you do not use this option, the conversion uses a default configuration. When you omit `-config`, to specify the properties of the input arguments, use `-args`.

`-globals` *`global_values`*	Specify names and initial values for global variables in MATLAB files.
	
	`global_values` is a cell array of global variable names and initial values. The format of `global_values` is:
	
	`{g1, init1, g2, init2, ..., gn, initn}`
	
	`gn` is the name of a global variable. `initn` is the initial value. For example:
	
	`-globals {'g', 5}`
	
	Alternatively, use this format:
	
	`-globals {global_var, {type, initial_value}}`
	
	`type` is a type object. To create the type object, use `coder.typeof`.
	
	If you do not provide initial values for global variables using the `-globals` option, `convertToSingle` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `convertToSingle` generates an error.

Version History

Introduced in R2015b

See Also

`coder.SingleConfig` | `coder.config`

Topics

"Generate Single-Precision MATLAB Code"

copyobj

Make independent copy of `quantizer` object

Syntax

```
q1 = copyobj(q)
[q1,q2,...] = copyobj(obja,objb,...)
```

Description

`q1 = copyobj(q)` makes a copy of `quantizer` object `q` and returns it in `q1`.

`[q1,q2,...] = copyobj(obja,objb,...)`copies `obja` into `q1`, `objb` into `q2`, and so on.

Using `copyobj` to copy a `quantizer` object is not the same as using the command syntax `q1 = q` to copy a `quantizer` object. `quantizer` objects have memory (their read-only properties). When you use `copyobj`, the resulting copy is independent of the original item; it does not share the original object's memory, such as the values of the properties `min`, `max`, `noverflows`, or `noperations`. Using `q1 = q` creates a new object that is an alias for the original and shares the original object's memory, and thus its property values.

Examples

```
q = quantizer([8 7]);
q1 = copyobj(q)
```

Version History

Introduced before R2006a

See Also

`quantizer` | `get` | `set`

cordicabs

CORDIC-based absolute value

Syntax

```
r = cordicabs(c)
r = cordicabs(c,niters)
r = cordicabs( ___ ,'ScaleOutput',b)
```

Description

`r = cordicabs(c)` returns the magnitude of the complex elements of `c`.

`r = cordicabs(c,niters)` performs the number of CORDIC algorithm iterations specified by `niters`.

`r = cordicabs( ___ ,'ScaleOutput',b)` specifies whether to scale the output by the inverse CORDIC gain factor.

Examples

CORDIC Absolute Value

Compare the absolute value computed using double-precision input values to the `cordicabs` and `abs` functions.

```
dblValues = complex(rand(5,4),rand(5,4))
```

```
dblValues = 5×4 complex

   0.8147 + 0.6557i   0.0975 + 0.7577i   0.1576 + 0.7060i   0.1419 + 0.8235i
   0.9058 + 0.0357i   0.2785 + 0.7431i   0.9706 + 0.0318i   0.4218 + 0.6948i
   0.1270 + 0.8491i   0.5469 + 0.3922i   0.9572 + 0.2769i   0.9157 + 0.3171i
   0.9134 + 0.9340i   0.9575 + 0.6555i   0.4854 + 0.0462i   0.7922 + 0.9502i
   0.6324 + 0.6787i   0.9649 + 0.1712i   0.8003 + 0.0971i   0.9595 + 0.0344i
```

```
r_dbl_ref = abs(dblValues)
```

```
r_dbl_ref = 5×4

    1.0458    0.7640    0.7234    0.8356
    0.9065    0.7936    0.9711    0.8128
    0.8586    0.6730    0.9964    0.9691
    1.3064    1.1604    0.4876    1.2371
    0.9277    0.9800    0.8062    0.9601
```

```
r_dbl_cdc = cordicabs(dblValues)
```

```
r_dbl_cdc = 5×4
```

```
    1.0458    0.7640    0.7234    0.8356
    0.9065    0.7936    0.9711    0.8128
    0.8586    0.6730    0.9964    0.9691
    1.3064    1.1604    0.4876    1.2371
    0.9277    0.9800    0.8062    0.9601
```

Compute absolute values of fixed-point inputs.

```
fxpValues = fi(dblValues)
```

```
fxpValues =
   0.8147 + 0.6557i   0.0975 + 0.7578i   0.1576 + 0.7061i   0.1419 + 0.8235i
   0.9058 + 0.0357i   0.2785 + 0.7431i   0.9706 + 0.0318i   0.4218 + 0.6948i
   0.1270 + 0.8491i   0.5469 + 0.3922i   0.9572 + 0.2769i   0.9157 + 0.3171i
   0.9134 + 0.9340i   0.9575 + 0.6555i   0.4854 + 0.0462i   0.7922 + 0.9502i
   0.6324 + 0.6787i   0.9649 + 0.1712i   0.8003 + 0.0971i   0.9595 + 0.0345i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
r_fxp_cdc = cordicabs(fxpValues)
```

```
r_fxp_cdc =
    1.0458    0.7640    0.7234    0.8356
    0.9066    0.7935    0.9712    0.8128
    0.8586    0.6730    0.9965    0.9691
    1.3064    1.1604    0.4876    1.2371
    0.9277    0.9799    0.8062    0.9601

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15
```

Input Arguments

c — Complex input array

scalar | vector | matrix | multidimensional array

Complex input array, specified as a scalar, vector, matrix, or multidimensional array. All input values must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`niters` — Number of iterations CORDIC algorithm performs

positive integer-valued scalar

Number of iterations the CORDIC algorithm performs, specified as a positive integer-valued scalar.

If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `c`. For floating-point operation, the maximum value is `52` for double or `23` for single.

Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

b — Whether to scale output by inverse CORDIC gain factor
1 (true) (default) | 0 (false)

Whether to scale output by inverse CORDIC gain factor, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Data Types: `logical`

Output Arguments

r — Magnitude values of the complex input values
scalar | vector | matrix | multidimensional array

Magnitude values of the complex input values, returned as a scalar, vector, matrix, or multidimensional array. If the input `c` is fixed point, then `r` is returned as a signed fixed-point data type using binary-point scaling. If the inputs are signed, then the word length of `r` is the input word length + 2. If the inputs are unsigned, then the word length of `r` is the input word length + 3. The fraction length of `r` is always the same as the fraction length of the input `c`.

More About

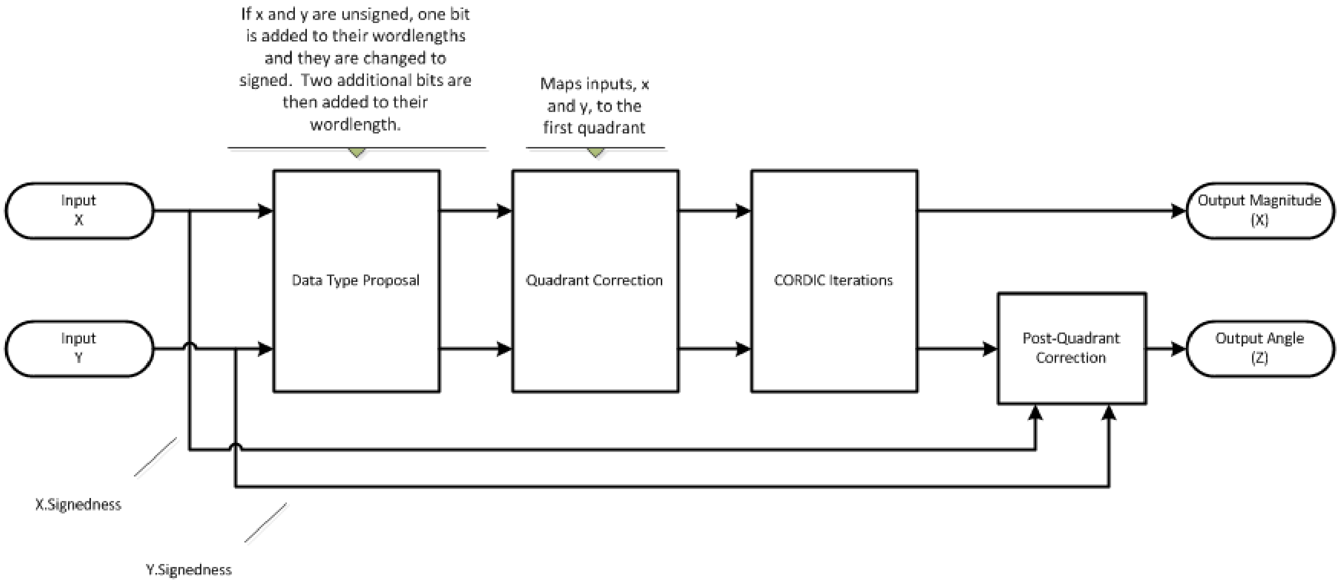
CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arcsine, arccosine, arctangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

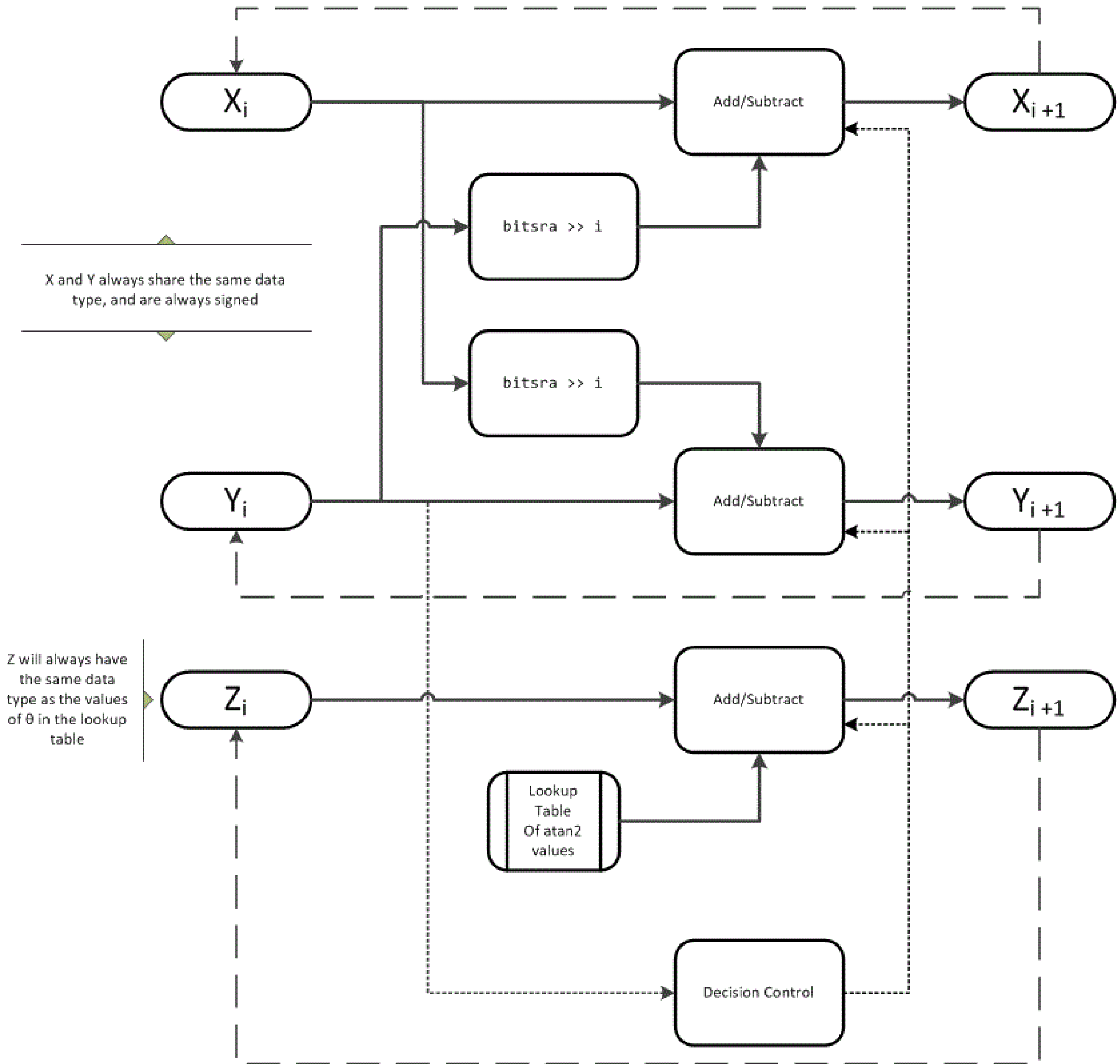
Increasing the number of CORDIC iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Algorithms

Signal Flow Diagrams



CORDIC Vectoring Kernel



The accuracy of the CORDIC kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

x_0 is initialized to the *x* input value

y_0 is initialized to the *y* input value

z_0 is initialized to 0

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2011b

References

[1] Volder, Jack E. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. EC-8, no. 3 (Sept. 1959): 330–334.

[2] Andraka, Ray. “A Survey of CORDIC Algorithm for FPGA Based Computers.” In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 191-200. https://dl.acm.org/doi/10.1145/275107.275139.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” In *Proceedings of the May 18-20, 1971 Spring Joint Computer Conference*, 379-386. https://dl.acm.org/doi/10.1145/1478786.1478840.

[4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*, no. 5 (May 1983): 317-325. https://doi.org/10.2307/2975781.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordiccart2pol` | `cordicangle` | `abs` | `abs`

cordicacos

CORDIC-based approximation of inverse cosine

Syntax

```
theta = cordicacos(x)
theta = cordicacos(x,niters)
```

Description

`theta = cordicacos(x)` returns the inverse cosine of `x` based on a double CORDIC (DCORDIC) algorithm approximation.

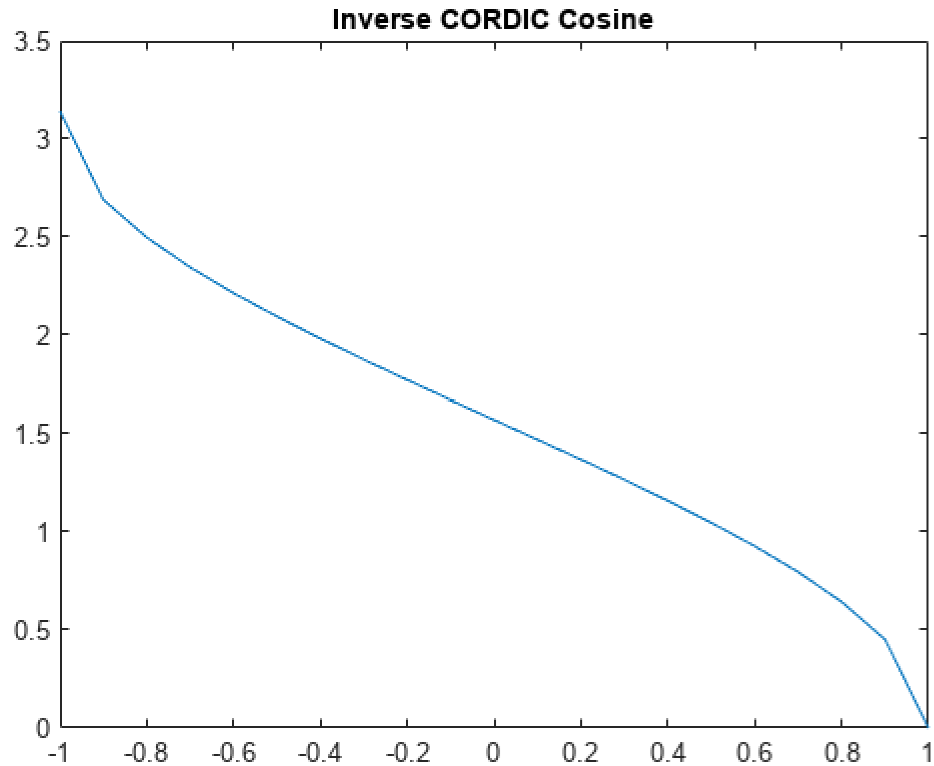
`theta = cordicacos(x,niters)` performs `niters` iterations of the DCORDIC algorithm.

Examples

Calculate CORDIC Inverse Cosine

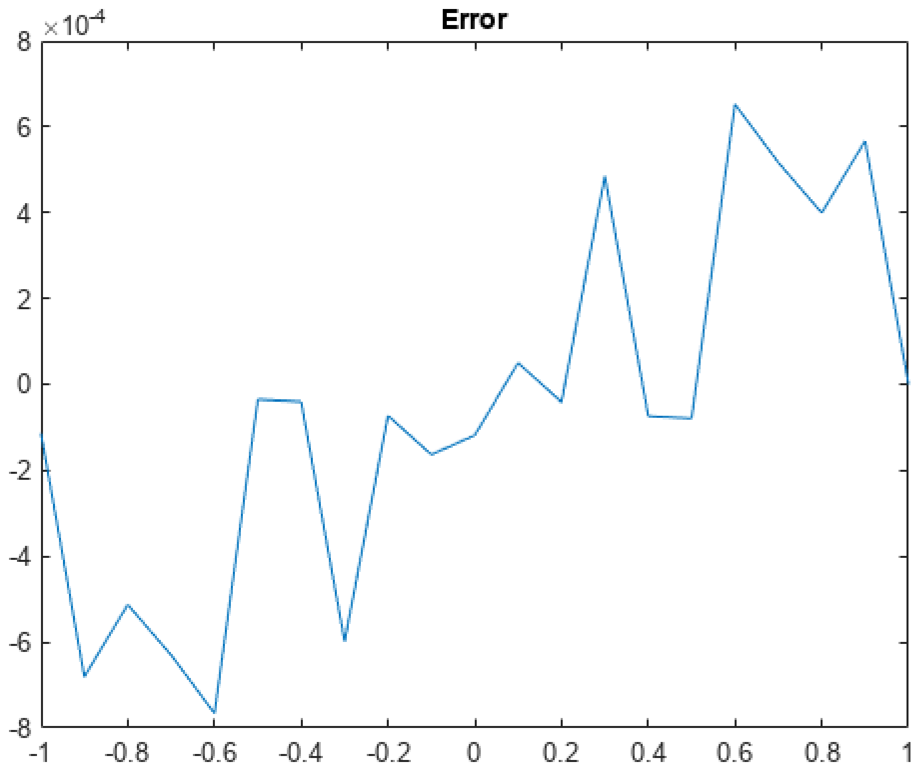
Compute the inverse cosine of a fixed-point `fi` object using a CORDIC implementation.

```
a = fi(-1:.1:1,1,16);
b = cordicacos(a);
plot(a,b);
title('Inverse CORDIC Cosine');
```



Compare the output of the `cordicacos` function and the `acos` function.

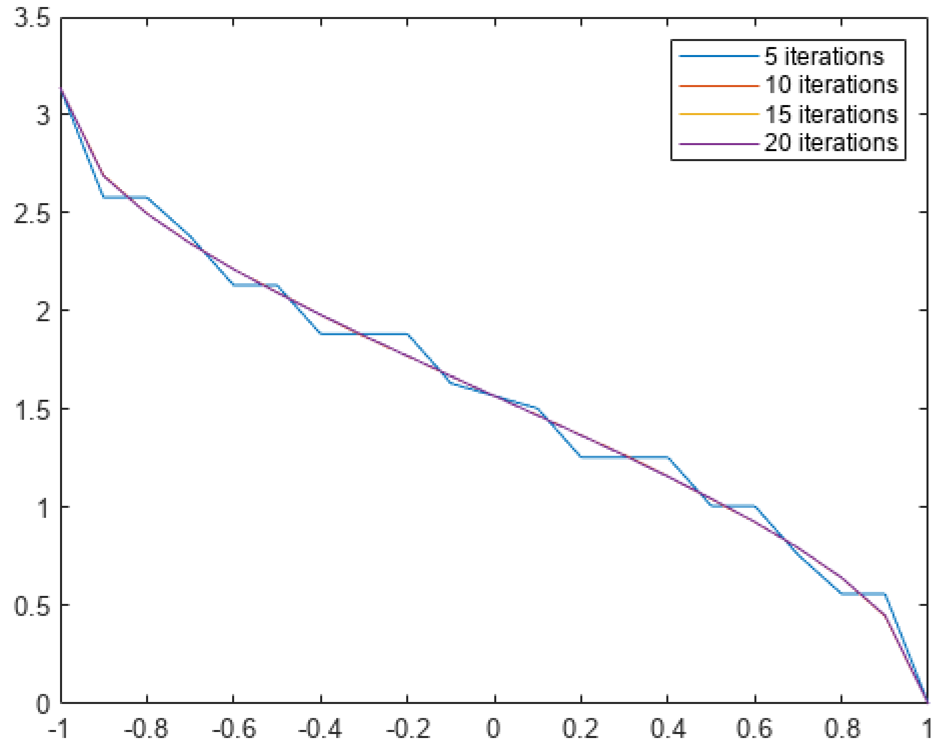
```
c = acos(double(a));
error = double(b)-c;
plot(a,error);
title('Error');
```



Calculate CORDIC Inverse Cosine with Specified Number of Iterations

Find the inverse cosine of a `fi` object using a CORDIC implementation and specify the number of iterations the CORDIC kernel should perform. Plot the CORDIC approximation of the inverse cosine with varying numbers of iterations.

```
a = fi(-1:.1:1,1,16);
for i = 5:5:20
    b = cordicacos(a,i);
    plot(a,b);
    hold on;
end
legend('5 iterations','10 iterations',...
    '15 iterations','20 iterations')
```



Input Arguments

x — Cosine of angle
scalar | vector | matrix | multidimensional array

Cosine of angle, specified as a scalar, vector, matrix, or multidimensional array. `x` must be a real, finite value in the range `[-1,1]`.

When `x` is specified as a `fi` object, the property `'DataTypeMode'` must equal `Fixed-point: binary point scaling`, `'Scaled double: binary point scaling'`, `'double'`, or `'single'`.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`niters` — Number of iterations CORDIC algorithm performs
positive integer-valued scalar

Number of iterations the CORDIC algorithm performs, specified as a positive integer-valued scalar.

If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is one less than the word length of the input array `x`. For double-precision inputs, the default value of `niters` is 54. For single-precision inputs, the default value is 25.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`theta` — Inverse cosine angle values
scalar | vector | matrix | multidimensional array

Inverse cosine angle values in radians, returned as a scalar, vector, matrix, or multidimensional array. The `cordicacos` function returns values in the interval `[0,π]`.

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arcsine, arccosine, arctangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Version History

Introduced in R2018b

References

[1] Volder, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. EC-8, no. 3 (Sept. 1959): 330–334.

[2] Andraka, Ray. "A Survey of CORDIC Algorithm for FPGA Based Computers." In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 191-200. https://dl.acm.org/doi/10.1145/275107.275139.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." In *Proceedings of the May 18-20, 1971 Spring Joint Computer Conference*, 379-386. https://dl.acm.org/doi/10.1145/1478786.1478840.

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*, no. 5 (May 1983): 317-325. https://doi.org/10.2307/2975781.

See Also

Functions
`cordicsin` | `cordiccos`

cordicangle

CORDIC-based phase angle

Syntax

```
theta = cordicangle(c)
theta = cordicangle(c,niters)
```

Description

`theta = cordicangle(c)` returns the phase angle in the interval `[-π,π]` for each element of a complex array `c`.

`theta = cordicangle(c,niters)` performs `niters` iterations of the CORDIC algorithm.

Examples

CORDIC-Based Phase Angle

Use the `cordicangle` function to compute the CORDIC-based phase angle for double-precision and fixed-point inputs.

```
dblRandomVals = complex(rand(5,4),rand(5,4));
theta_dbl_ref = angle(dblRandomVals);
theta_dbl_cdc = cordicangle(dblRandomVals)
```

```
theta_dbl_cdc = 5×4

    0.6777    1.4428    1.3512    1.4002
    0.0394    1.2122    0.0328    1.0252
    1.4223    0.6222    0.2816    0.3334
    0.7966    0.6003    0.0948    0.8758
    0.8208    0.1756    0.1208    0.0359
```

```
fxpRandomVals = fi(dblRandomVals);
theta_fxp_cdc = cordicangle(fxpRandomVals)
```

```
theta_fxp_cdc = 
    0.6777    1.4426    1.3513    1.4001
    0.0393    1.2122    0.0327    1.0254
    1.4224    0.6223    0.2817    0.3333
    0.7964    0.6003    0.0950    0.8757
    0.8208    0.1758    0.1208    0.0359

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

c — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array. When the elements of `c` are nonnegative real numbers, `cordicangle` returns `0`. When the elements of `c` are negative real numbers, `cordicangle` returns `π`.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`niters` — Number of iterations CORDIC algorithm performs
positive integer-valued scalar

Number of iterations the CORDIC algorithm performs, specified as a positive integer-valued scalar.

Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

If you do not specify `niters`, or if you specify it as empty or nonfinite, the maximum allowed value is used. For fixed-point input, the maximum number of iterations is one less than the word length of `c`. For double-precision input, the maximum value is 52. For single-precision input, the maximum value is 23.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`theta` — Phase angle in radians
scalar | vector | matrix | multidimensional array

Phase angle in radians, returned as a scalar, vector, matrix, or multidimensional array.

If `c` is floating-point, then `theta` has the same data type as `c`. Otherwise, `theta` has a fixed-point data type with the same word length as `c` and with a best-precision fraction length.

More About

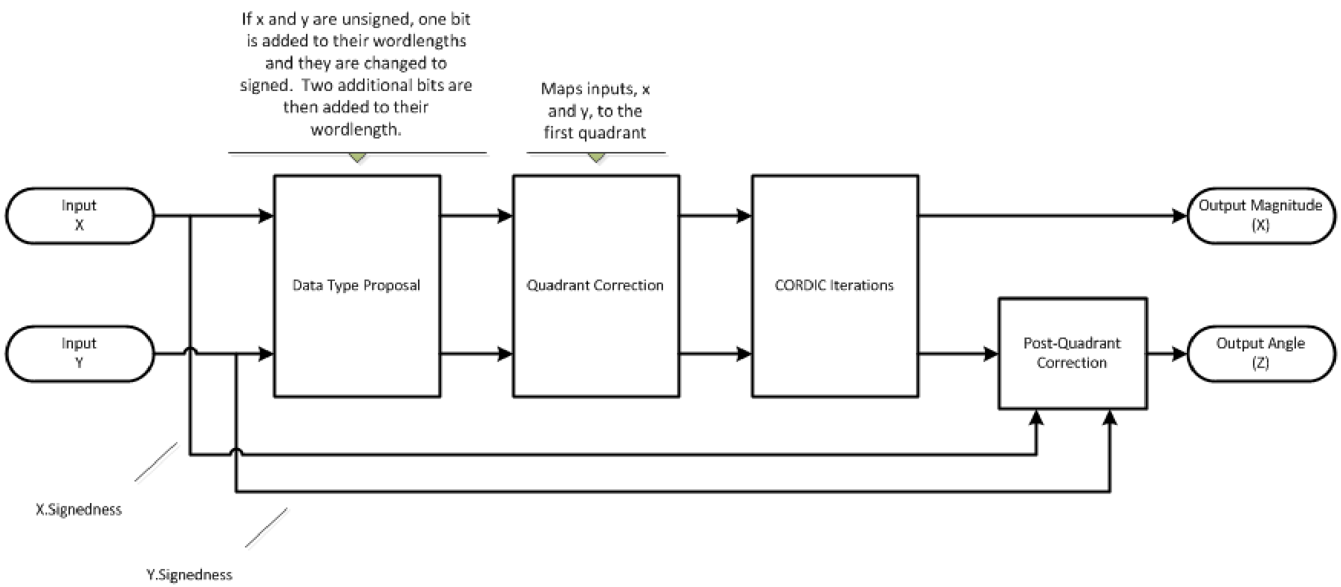
CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arcsine, arccosine, arctangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

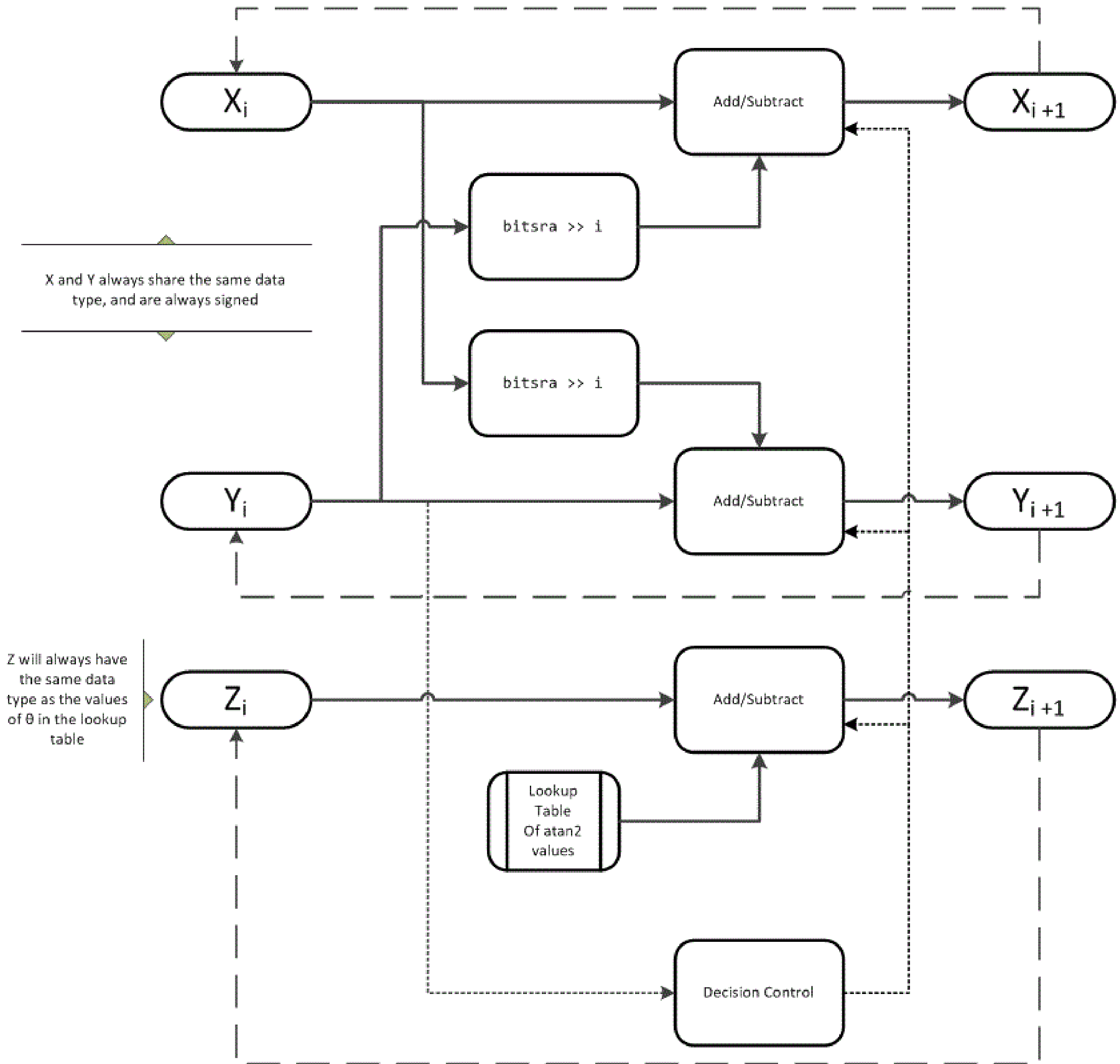
Increasing the number of CORDIC iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Algorithms

Signal Flow Diagrams



CORDIC Vectoring Kernel



The accuracy of the CORDIC kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

x_0 is initialized to the *x* input value

y_0 is initialized to the *y* input value

z_0 is initialized to 0

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2011b

References

[1] Volder, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. EC-8, no. 3 (Sept. 1959): 330–334.

[2] Andraka, Ray. "A Survey of CORDIC Algorithm for FPGA Based Computers." In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 191-200. https://dl.acm.org/doi/10.1145/275107.275139.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." In *Proceedings of the May 18-20, 1971 Spring Joint Computer Conference*, 379-386. https://dl.acm.org/doi/10.1145/1478786.1478840.

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*, no. 5 (May 1983): 317-325. https://doi.org/10.2307/2975781.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordicatan2` | `cordiccart2pol` | `cordicabs` | `angle`

cordicasin

CORDIC-based approximation of inverse sine

Syntax

```
theta = cordicasin(x)
theta = cordicasin(x, niters)
```

Description

`theta = cordicasin(x)` returns the inverse sine of `x` based on a CORDIC approximation.

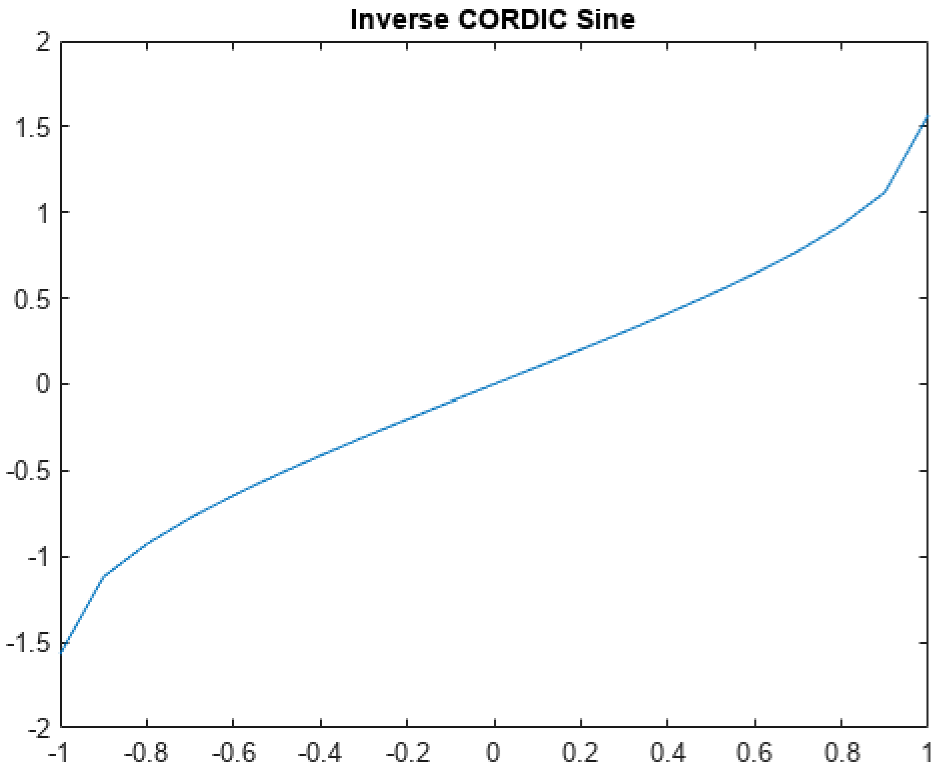
`theta = cordicasin(x, niters)` returns the inverse sine of `x` performing `niters` iterations of the CORDIC algorithm.

Examples

Calculate CORDIC Inverse Sine

Compute the inverse Sine of a fixed-point `fi` object using a CORDIC implementation.

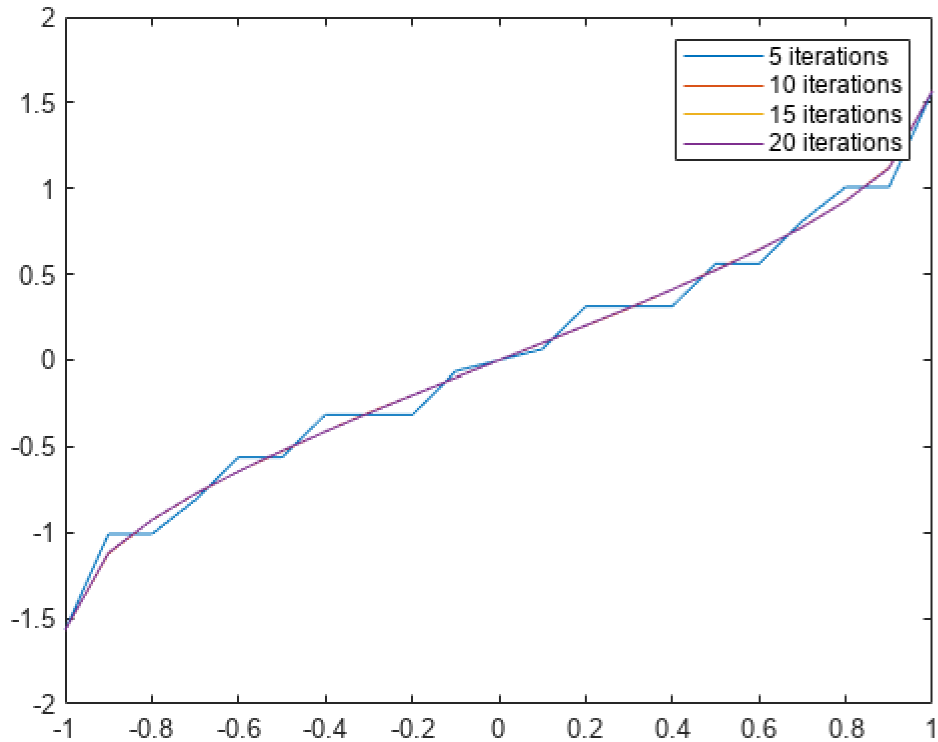
```
a = fi(-1:.1:1,1,16);
b = cordicasin(a);
plot(a, b);
title('Inverse CORDIC Sine');
```



Calculate CORDIC Inverse Sine with Specified Number of Iterations

Find the inverse sine of a `fi` object using a CORDIC implementation and specify the number of iterations the CORDIC kernel should perform. Plot the CORDIC approximation of the inverse sine with varying numbers of iterations.

```
a = fi(-1:.1:1,1,16);
for i = 5:5:20
    b = cordicasin(a,i);
    plot(a,b);
    hold on;
end
legend('5 iterations','10 iterations',...
    '15 iterations','20 iterations')
```



Input Arguments

x — Numeric input

scalar | vector | matrix | multidimensional array

Numeric input, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`niters` — Number of iterations

scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is one less than the word length of the input array, `theta`. For double-precision inputs, the default value of `niters` is 52. For single-precision inputs, the default value is 23.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`theta` — Inverse sine angle values
scalar | vector | matrix | n-dimensional array

Inverse sine angle values in rad.

Version History

Introduced in R2018b

See Also

Functions
`cordicsin` | `cordiccos`

cordicatan2

CORDIC-based four quadrant inverse tangent

Syntax

```
theta = cordicatan2(y,x)
theta = cordicatan2(y,x,niters)
```

Description

`theta = cordicatan2(y,x)` computes the four quadrant arctangent of `y` and `x` using a CORDIC algorithm approximation.

`theta = cordicatan2(y,x,niters)` performs `niters` iterations of the algorithm.

Examples

Compute CORDIC Arctangent

Define floating-point Cartesian coordinates.

```
y = 0.5;
x = -0.5;
```

Use `cordicatan2` to compute floating-point CORDIC arctangent. Compare the result to the arctangent computed using `atan2`.

```
theta_cdat2_float = cordicatan2(y,x)
```

```
theta_cdat2_float = 2.3562
```

```
theta_atan2_float = atan2(y,x)
```

```
theta_atan2_float = 2.3562
```

Define fixed-point Cartesian coordinates.

```
y = fi(0.5,1,16,15);
x = fi(-0.5,1,16,15);
```

Use `cordicatan2` to compute fixed-point CORDIC arctangent. Compare the result to the arctangent computed using `atan2`.

```
theta_cdat2_fixpt = cordicatan2(y,x)
```

```
theta_cdat2_fixpt = 
    2.3562

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
theta_atan2_fixpt = atan2(y,x)

theta_atan2_fixpt = 
    2.3562

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

y — Cartesian *y*-coordinate
scalar | vector | matrix | multidimensional array

Cartesian *y*-coordinate, specified as a scalar, vector, matrix, or multidimensional array.

`y` and `x` must be the same size. If they are not the same size, at least one value must be a scalar value. `y` and `x` must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

x — Cartesian *x*-coordinate
scalar | vector | matrix | multidimensional array

Cartesian *x*-coordinate, specified as a scalar, vector, matrix, or multidimensional array.

`y` and `x` must be the same size. If they are not the same size, at least one value must be a scalar value. `y` and `x` must have the same data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`niters` — Number of iterations of CORDIC algorithm
positive integer-valued scalar

Number of iterations of CORDIC algorithm, specified as a positive, integer-valued scalar.

Increasing the number of iterations can produce more accurate results, but also increases the expense of the computation and adds latency.

If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `y` or `x`. For floating-point operation, the maximum value is 52 for double or 23 for single.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`theta` — Arctangent value
scalar | vector | matrix | multidimensional array

Arctangent value in the range [-pi, pi] radians, returned as a scalar, vector, matrix, or multidimensional array.

If `y` and `x` are floating-point numbers, then `theta` has the same data type as `y` and `x`. Otherwise, `theta` is a fixed-point data type with the same word length as `y` and `x` and with a best-precision fraction length for the [-pi, pi] range.

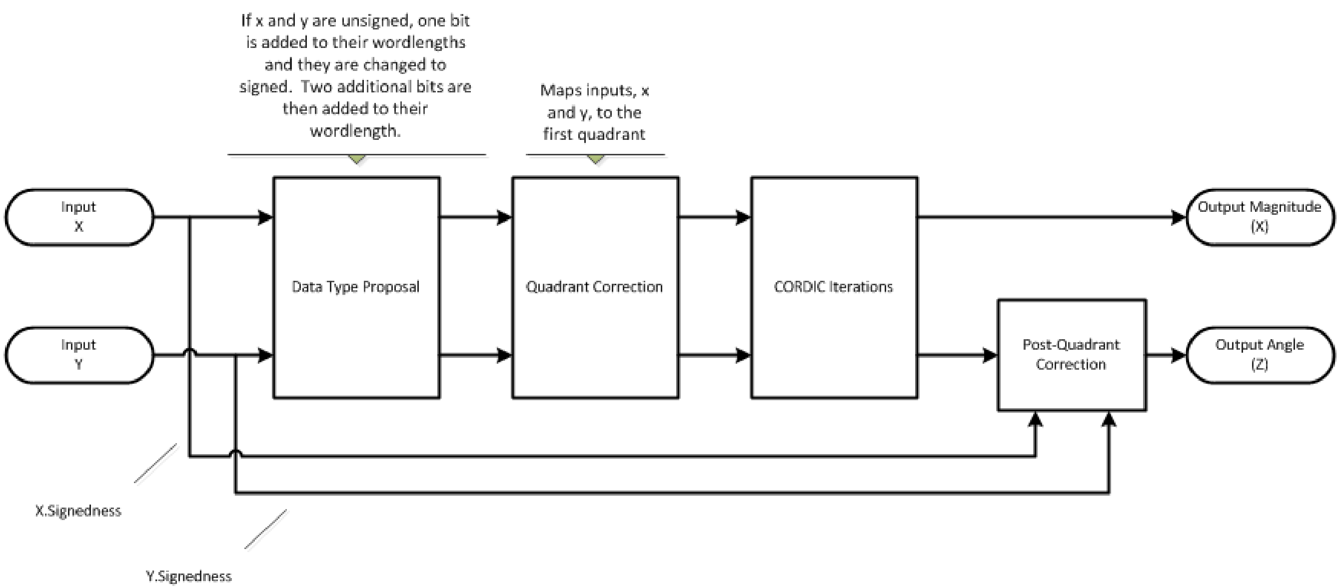
Algorithms

CORDIC

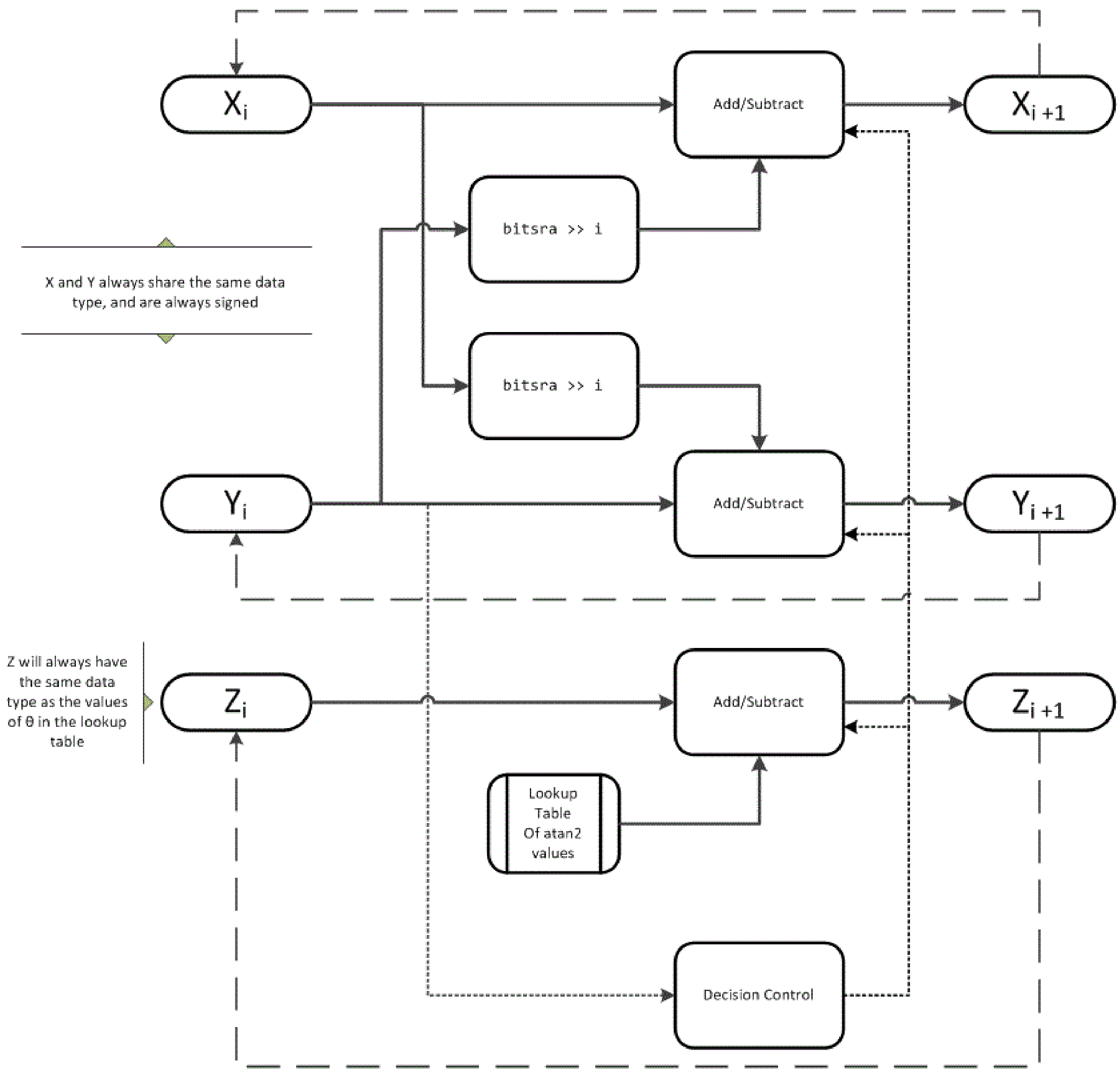
CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Signal Flow Diagram



CORDIC Vectoring Kernel



The accuracy of the CORDIC kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

- X_0 is initialized to the *X* input value
- Y_0 is initialized to the *Y* input value
- Z_0 is initialized `0`

`fimath` Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction` — `Wrap`
- `RoundingMethod` — `Floor`

The output has no attached `fimath`.

Version History

Introduced in R2011b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`atan2` | `atan2` | `cordicsin` | `cordiccos`

cordiccart2pol

CORDIC-based approximation of Cartesian-to-polar conversion

Syntax

```
[theta,r] = cordiccart2pol(x,y)
[theta,r] = cordiccart2pol(x,y, niters)
[theta,r] = cordiccart2pol(x,y, niters,'ScaleOutput',b)
[theta,r] = cordiccart2pol(x,y, 'ScaleOutput',b)
```

Description

`[theta,r] = cordiccart2pol(x,y)` using a CORDIC algorithm approximation, returns the polar coordinates, angle `theta` and radius `r`, of the Cartesian coordinates, `x` and `y`.

`[theta,r] = cordiccart2pol(x,y, niters)` performs `niters` iterations of the algorithm.

`[theta,r] = cordiccart2pol(x,y, niters,'ScaleOutput',b)` specifies both the number of iterations and, depending on the Boolean value of `b`, whether to scale the `r` output by the inverse CORDIC gain value.

`[theta,r] = cordiccart2pol(x,y, 'ScaleOutput',b)` scales the `r` output by the inverse CORDIC gain value, depending on the Boolean value of `b`.

Input Arguments

x,y

`x,y` are Cartesian coordinates. `x` and `y` must be the same size. If they are not the same size, at least one value must be a scalar value. Both `x` and `y` must have the same data type.

niters

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

ScaleOutput

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Default: `true`

Output Arguments

theta

theta contains the polar coordinates angle values, which are in the range [–pi, pi] radians. If x and y are floating-point, then theta has the same data type as x and y. Otherwise, theta is a fixed-point data type with the same word length as x and y and with a best-precision fraction length for the [-pi, pi] range.

r

r contains the polar coordinates radius magnitude values. r is real-valued and can be a scalar value or have the same dimensions as theta If the inputs x,y are fixed-point values, r is also fixed point (and is always signed, with binary point scaling). Both x,y input values must have the same data type. If the inputs are signed, then the word length of r is the input word length + 2. If the inputs are unsigned, then the word length of r is the input word length + 3. The fraction length of r is always the same as the fraction length of the x,y inputs.

Examples

Convert fixed-point Cartesian coordinates to polar coordinates.

```
[thPos,r]=cordiccart2pol(sfi([0.75:-0.25:-1.0],16,15),sfi(0.5,16,15))

thPos =

    0.5881  0.7854  1.1072  1.5708  2.0344  2.3562  2.5535  2.6780

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

r =

    0.9014  0.7071  0.5591  0.5000  0.5591  0.7071  0.9014  1.1180

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15

[thNeg,r]=...
  cordiccart2pol(sfi([0.75:-0.25:-1.0],16,15),sfi(-0.5,16,15))

thNeg =

 -0.5881 -0.7854 -1.1072 -1.5708 -2.0344 -2.3562 -2.5535 -2.6780

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

r =
```

```
 0.9014  0.7071  0.5591  0.5000  0.5591  0.7071  0.9014  1.1180

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15
```

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

More About

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

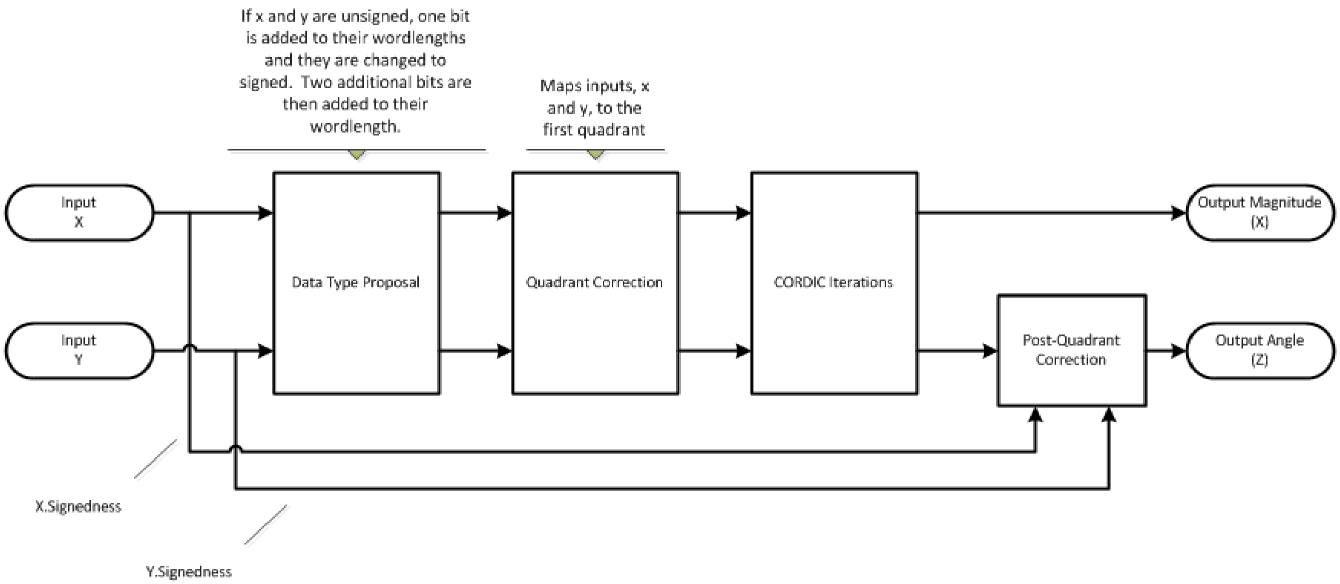
[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

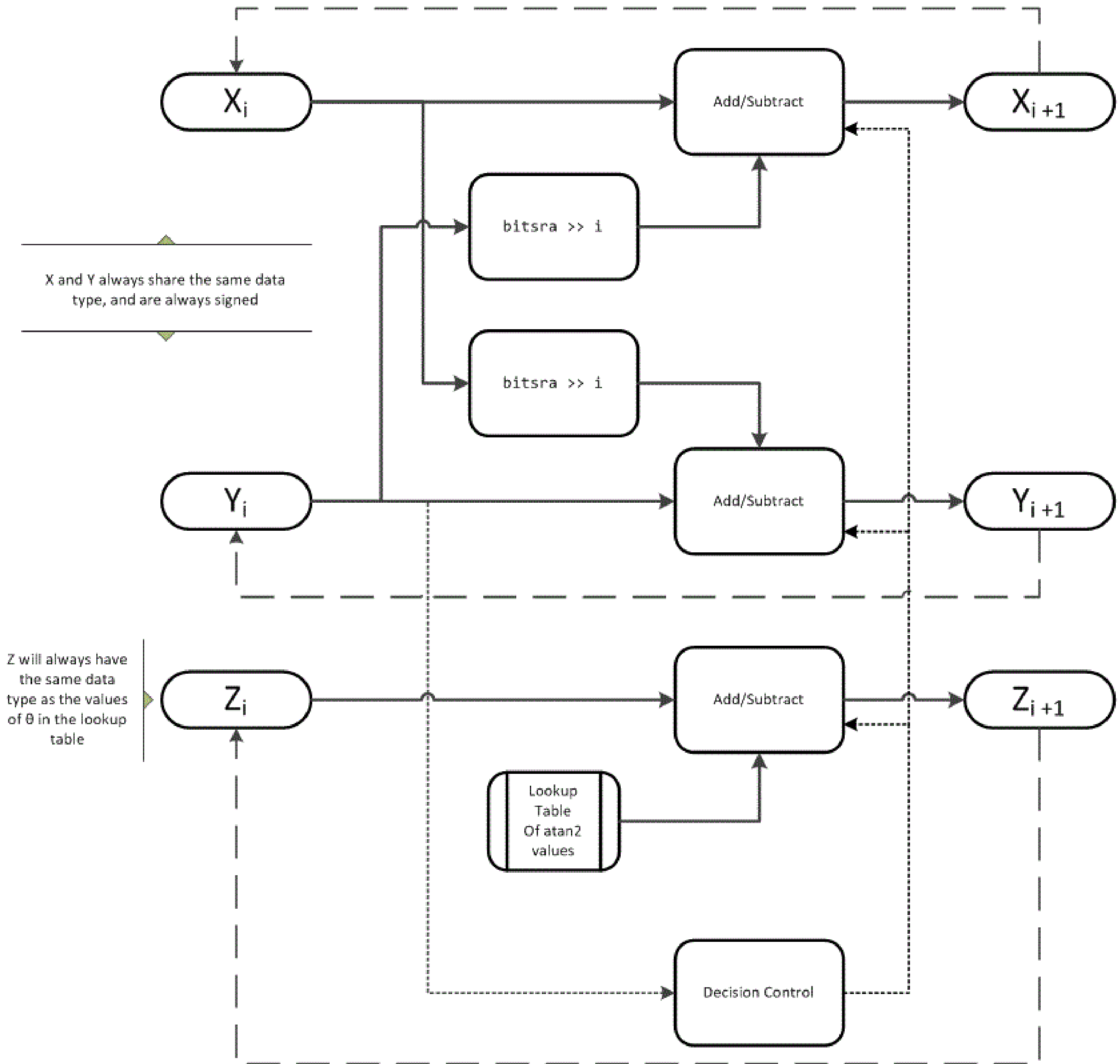
[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Algorithms

Signal Flow Diagrams



CORDIC Vectoring Kernel



The accuracy of the CORDIC kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

x_0 is initialized to the x input value

y_0 is initialized to the y input value

z_0 is initialized to 0

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2011b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordicatan2` | `cordicpol2cart` | `cart2pol`

cordiccexp

CORDIC-based approximation of complex exponential

Syntax

```
y = cordiccexp(theta,niters)
```

Description

`y = cordiccexp(theta,niters)` computes `cos(theta)` + *j**`sin(theta)` using a "CORDIC" on page 4-285 algorithm approximation. *y* contains the approximated complex result.

Input Arguments

`theta`

`theta` can be a signed or unsigned scalar, vector, matrix, or `N`-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range [–2π 2π).

`niters`

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

`y`

`y` is the approximated complex result of the `cordiccexp` function. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength – 2`.

Examples

The following example illustrates the effect of the number of iterations on the result of the `cordiccexp` approximation.

```
wrdLn = 8;
theta = fi(pi/2, 1, wrdLn);
fprintf('\n\nNITERS\t\tY (SIN)\t ERROR\t LSBs\t\tX (COS)\t ERROR\t LSBs\n');
fprintf('------\t\t-------\t ------\t ----\t\t-------\t ------\t ----\n');
for niters = 1:(wrdLn - 1)
 cis    = cordiccexp(theta, niters);
 fl     = cis.FractionLength;
 x      = real(cis);
 y      = imag(cis);
 x_dbl  = double(x);
 x_err  = abs(x_dbl - cos(double(theta)));
 y_dbl  = double(y);
 y_err  = abs(y_dbl - sin(double(theta)));
 fprintf('%d\t\t%1.4f\t%1.4f\t%1.1f\t\t%1.4f\t%1.4f\t%1.1f\n',...
   niters,y_dbl,y_err,(y_err*pow2(fl)),x_dbl,x_err,(x_err*pow2(fl)));
end
fprintf('\n');
```

The output table appears as follows:

```
NITERS   Y (SIN)  ERROR   LSBs    X (COS)  ERROR   LSBs
------    -------  ------  ----    -------  ------  ----
1         0.7031   0.2968  19.0     0.7031   0.7105  45.5
2         0.9375   0.0625  4.0      0.3125   0.3198  20.5
3         0.9844   0.0156  1.0      0.0938   0.1011  6.5
4         0.9844   0.0156  1.0      -0.0156  0.0083  0.5
5         1.0000   0.0000  0.0      0.0312   0.0386  2.5
6         1.0000   0.0000  0.0      0.0000   0.0073  0.5
7         1.0000   0.0000  0.0      0.0156   0.0230  1.5
```

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

More About

[1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

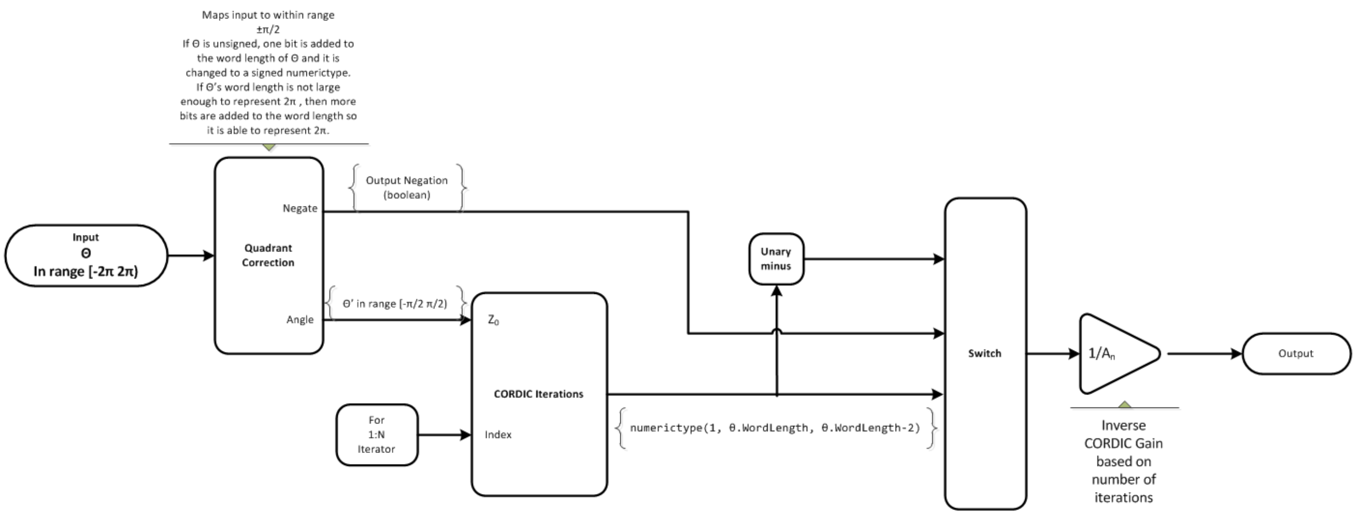
[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

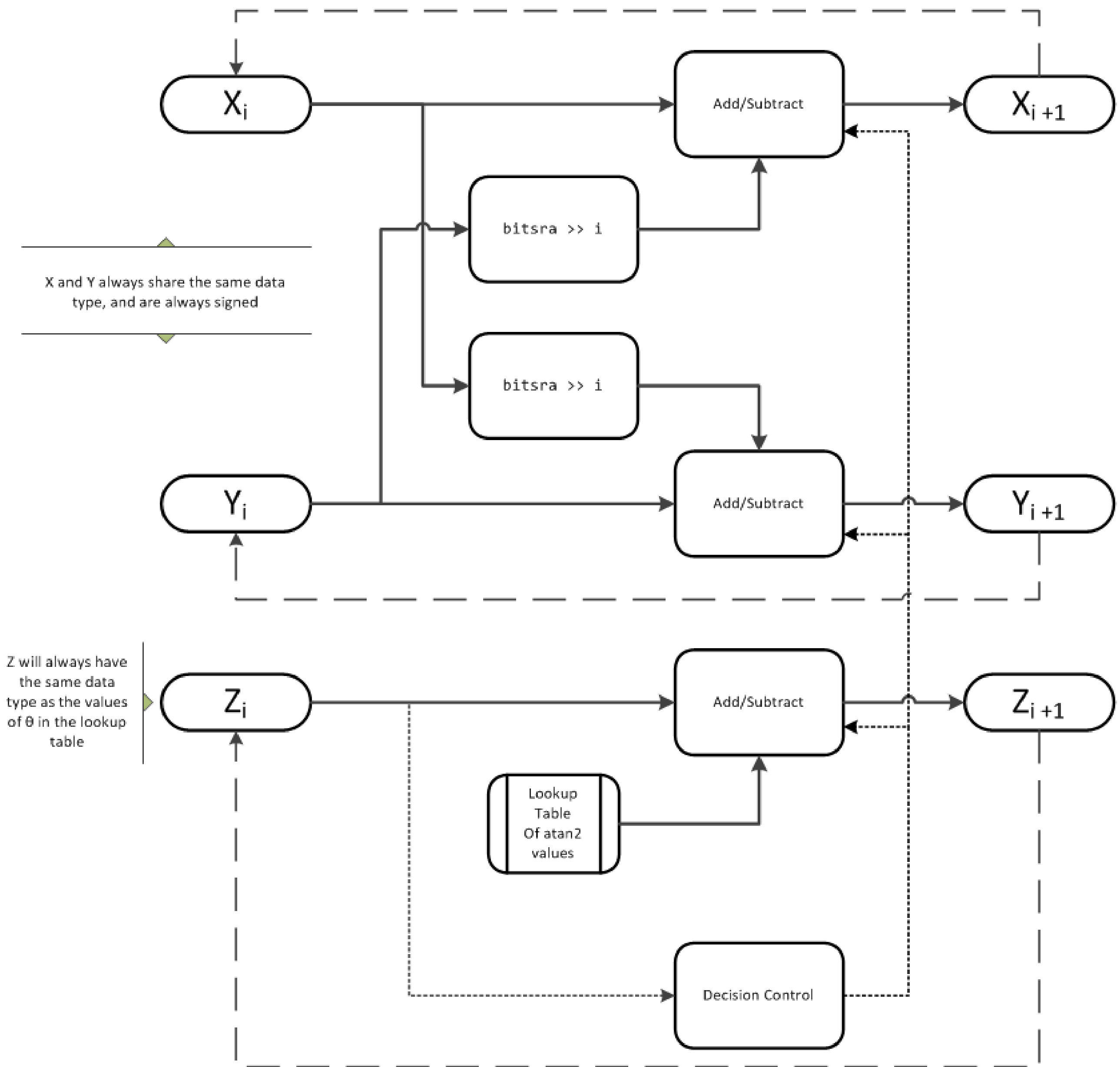
[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Algorithms

Signal Flow Diagrams



CORDIC Rotation Kernel



X represents the real part, *Y* represents the imaginary part, and *Z* represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

z_0 is initialized to the θ input argument value

x_0 is initialized to $\dfrac{1}{A_N}$

y_0 is initialized to 0

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordiccos` | `cordicsin` | `cordicsincos`

cordiccos

CORDIC-based approximation of cosine

Syntax

`y = cordiccos(theta, niters)`

Description

`y = cordiccos(theta, niters)` computes the cosine of *theta* using a "CORDIC" on page 4-291 algorithm approximation.

Input Arguments

theta

`theta` can be a signed or unsigned scalar, vector, matrix, or `N`-dimensional array containing the angle values in radians. All values of `theta` must be real and in the range [–2π 2π).

niters

`niters` is the number of iterations the CORDIC algorithm performs. This is an optional argument. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

y

y is the CORDIC-based approximation of the cosine of `theta`. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength – 2`.

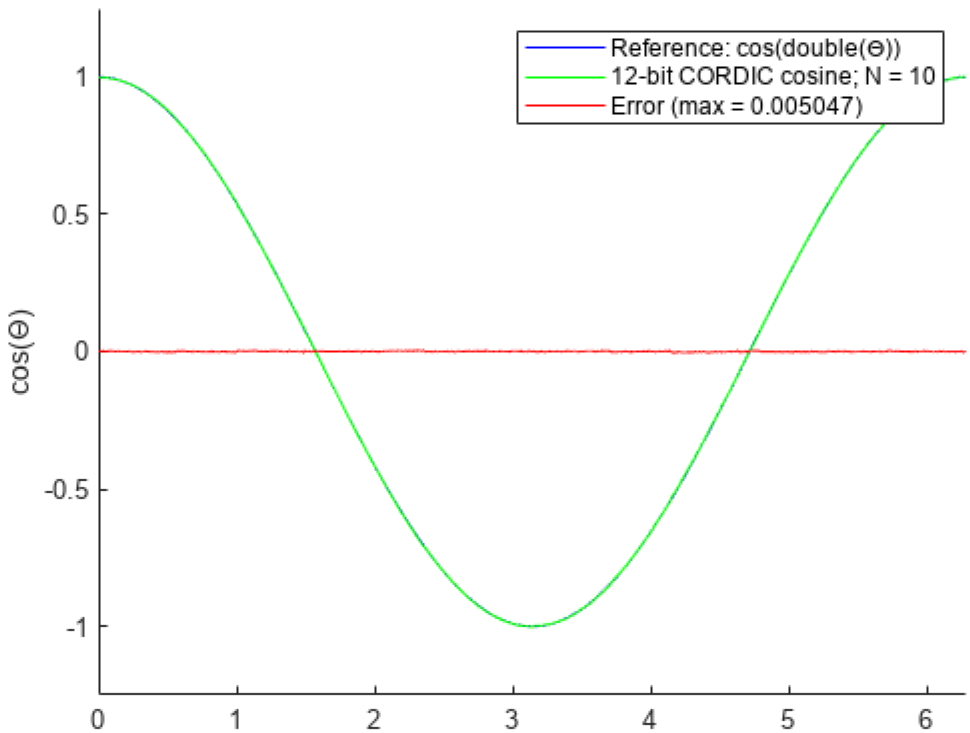
Examples

Compare Results of cordiccos and cos Functions

Compare the results produced by various iterations of the `cordiccos` algorithm to the results of the double-precision `cos` function.

```
% Create 1024 points between [0,2*pi)
stepSize = pi/512;
thRadDbl = 0:stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDbl,12);    % signed, 12-bit fixed-point
```

```
cosThRef = cos(double(thRadFxp));   % reference results

% Use 12-bit quantized inputs and vary the number
% of iterations from 2 to 10.
% Compare the  fixed-point CORDIC results to the
% double-precision trig function results.
for niters = 2:2:10
    cdcCosTh  = cordiccos(thRadFxp,niters);
    errCdcRef = cosThRef - double(cdcCosTh);
end

figure
hold on
axis([0 2*pi -1.25 1.25]);
    plot(thRadFxp,cosThRef,'b');
    plot(thRadFxp,cdcCosTh,'g');
    plot(thRadFxp,errCdcRef,'r');
    ylabel('cos(\Theta)');
    gca.XTick = 0:pi/2:2*pi;
    gca.XTickLabel = {'0','pi/2','pi','3*pi/2','2*pi'};
    gca.YTick = -1:0.5:1;
    gca.YTickLabel = {'-1.0','-0.5','0','0.5','1.0'};
    ref_str = 'Reference: cos(double(\Theta))';
    cdc_str = sprintf('12-bit CORDIC cosine; N = %d',niters);
    err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
    legend(ref_str,cdc_str,err_str);
```



After 10 iterations, the CORDIC algorithm has approximated the cosine of `theta` to within 0.005187 of the double-precision cosine result.

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

More About

[1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

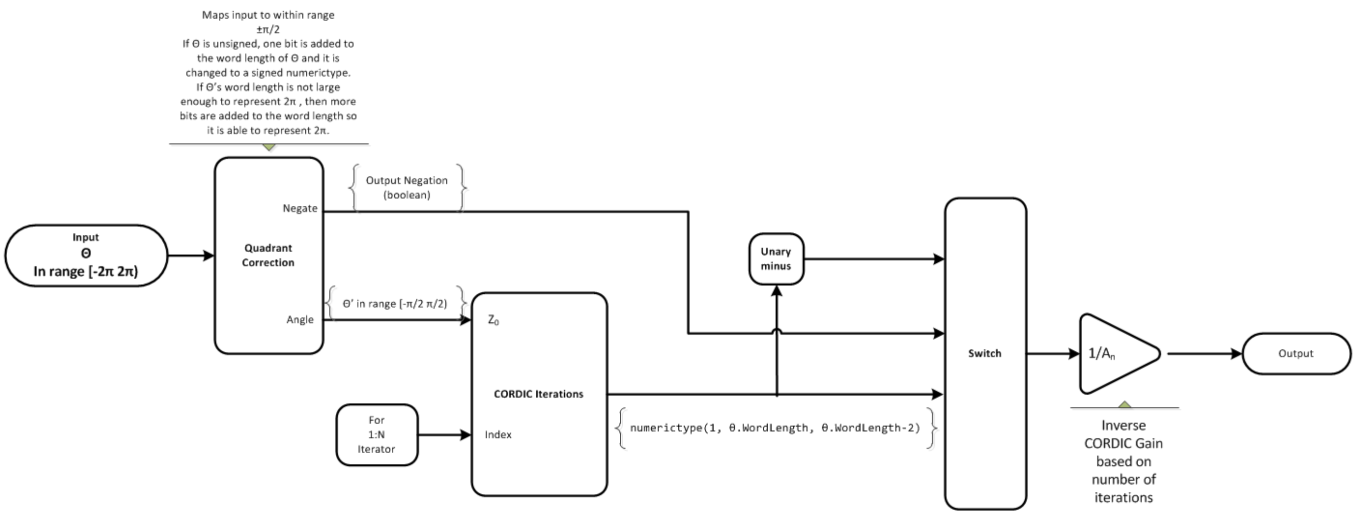
[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

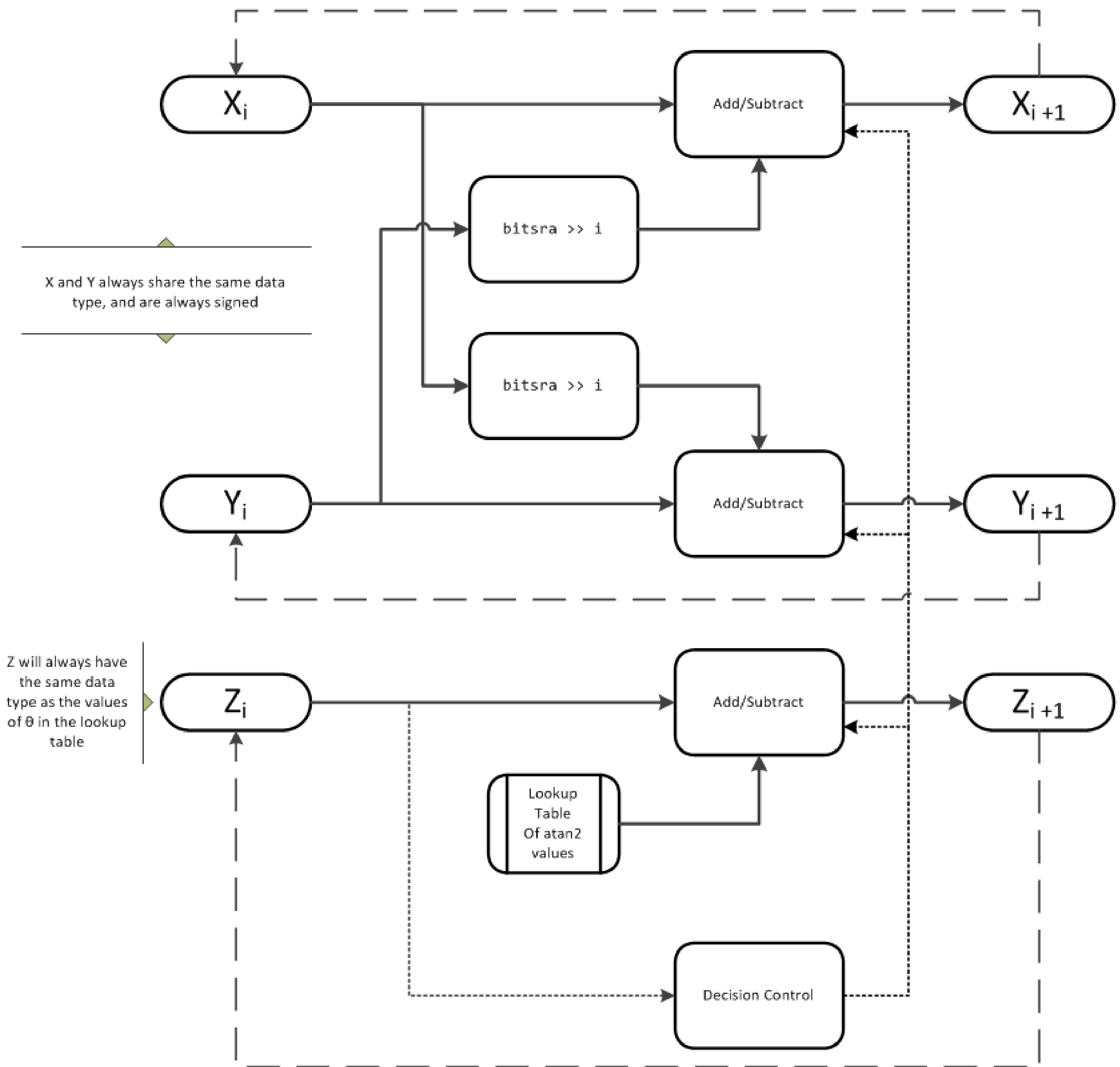
[4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Algorithms

Signal Flow Diagrams



CORDIC Rotation Kernel



X represents the sine, *Y* represents the cosine, and *Z* represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

> z_0 is initialized to the θ input argument value
>
> x_0 is initialized to $\frac{1}{A_N}$
>
> y_0 is initialized to 0

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

You can generate HDL code for `cordiccos` function.

See Also

`cordiccexp` | `cordicsin` | `cordicsincos` | `sin` | `cos`

cordicpol2cart

CORDIC-based approximation of polar-to-Cartesian conversion

Syntax

```
[x,y] = cordicpol2cart(theta,r)
[x,y] = cordicpol2cart(theta,r,niters)
[x,y] = cordicpol2cart(theta,r,Name,Value)
[x,y] = cordicpol2cart(theta,r,niters,Name,Value)
```

Description

`[x,y] = cordicpol2cart(theta,r)` returns the Cartesian xy coordinates of `r`* e^(j*`theta`) using a CORDIC algorithm approximation.

`[x,y] = cordicpol2cart(theta,r,niters)` performs `niters` iterations of the algorithm.

`[x,y] = cordicpol2cart(theta,r,Name,Value)` scales the output depending on the Boolean value of `b`.

`[x,y] = cordicpol2cart(theta,r,niters,Name,Value)` specifies both the number of iterations and `Name,Value` pair for whether to scale the output.

Input Arguments

`theta`

`theta` can be a signed or unsigned scalar, vector, matrix, or *N*-dimensional array containing the angle values in radians. All values of `theta` must be in the range [–2π 2π).

`r`

`r` contains the input magnitude values and can be a scalar or have the same dimensions as `theta`. `r` must be real valued.

`niters`

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `r` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

`ScaleOutput`

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Default: true

Output Arguments

`[x,y]`

`[x,y]` contains the approximated Cartesian coordinates. When the input `r` is floating point, the output `[x,y]` has the same data type as the input.

When the input `r` is a *signed* integer or fixed point data type, the outputs `[x,y]` are signed `fi` objects. These `fi` objects have word lengths that are two bits larger than that of `r`. Their fraction lengths are the same as the fraction length of `r`.

When the input `r` is an *unsigned* integer or fixed point, the outputs `[x,y]` are signed `fi` objects. These `fi` objects have word lengths are three bits larger than that of `r`. Their fraction lengths are the same as the fraction length of `r`.

Examples

Run the following code, and evaluate the accuracy of the CORDIC-based Polar-to-Cartesian conversion.

```
wrdLn = 16;
theta = fi(pi/3, 1, wrdLn);
u     = fi( 2.0, 1, wrdLn);

fprintf('\n\nNITERS\tX\t\t ERROR\t LSBs\t\tY\t\t ERROR\t LSBs\n');
fprintf('------\t-------\t ------\t ----\t\t-------\t ------\t ----\n');
for niters = 1:(wrdLn - 1)
 [x_ref, y_ref] = pol2cart(double(theta),double(u));
 [x_fi,  y_fi] = cordicpol2cart(theta, u, niters);
 x_dbl  = double(x_fi);
 y_dbl  = double(y_fi);
 x_err  = abs(x_dbl - x_ref);
 y_err  = abs(y_dbl - y_ref);
 fprintf('%d\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
   niters,x_dbl,x_err,(x_err * pow2(x_fi.FractionLength)),...
   y_dbl,y_err,(y_err * pow2(y_fi.FractionLength)));
end
fprintf('\n');
```

```
NITERS  X        ERROR    LSBs      Y      ERROR   LSBs
------    -------     ------     ----        -------     ------     ----
   1      1.4142   0.4142   3392.8  1.4142   0.3178   2603.8
   2      0.6324   0.3676   3011.2  1.8973   0.1653   1354.2
   3      1.0737   0.0737   603.8   1.6873   0.0448   366.8
   4      0.8561   0.1440   1179.2  1.8074   0.0753   617.2
   5      0.9672   0.0329   269.2   1.7505   0.0185   151.2
   6      1.0214   0.0213   174.8   1.7195   0.0126   102.8
   7      0.9944   0.0056   46.2    1.7351   0.0031   25.2
   8      1.0079   0.0079   64.8    1.7274   0.0046   37.8
   9      1.0011   0.0011   8.8     1.7313   0.0007   5.8
   10   0.9978   0.0022   18.2    1.7333   0.0012   10.2
   11   0.9994   0.0006   5.2     1.7323   0.0003   2.2
   12   1.0002   0.0002   1.8     1.7318   0.0002   1.8
   13   0.9999   0.0002   1.2     1.7321   0.0000   0.2
   14   0.9996   0.0004   3.2     1.7321   0.0000   0.2
   15   0.9998   0.0003   2.2     1.7321   0.0000   0.2
```

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

More About

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

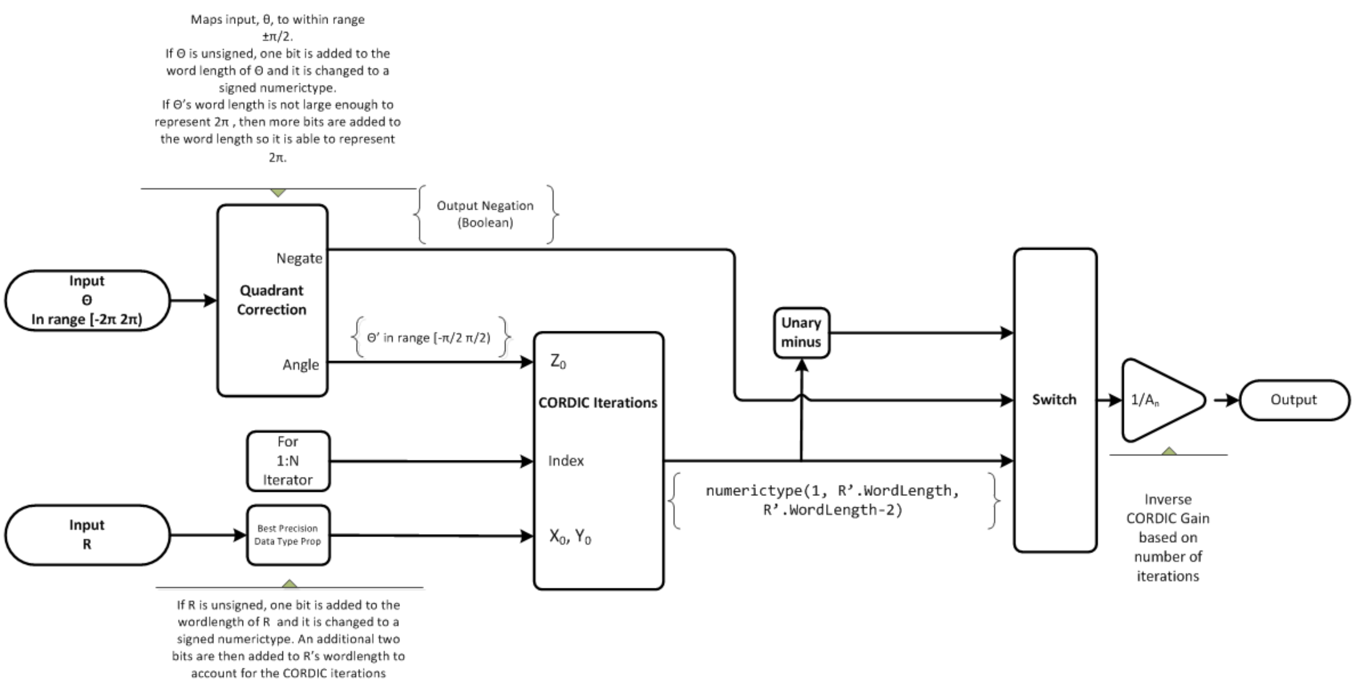
[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

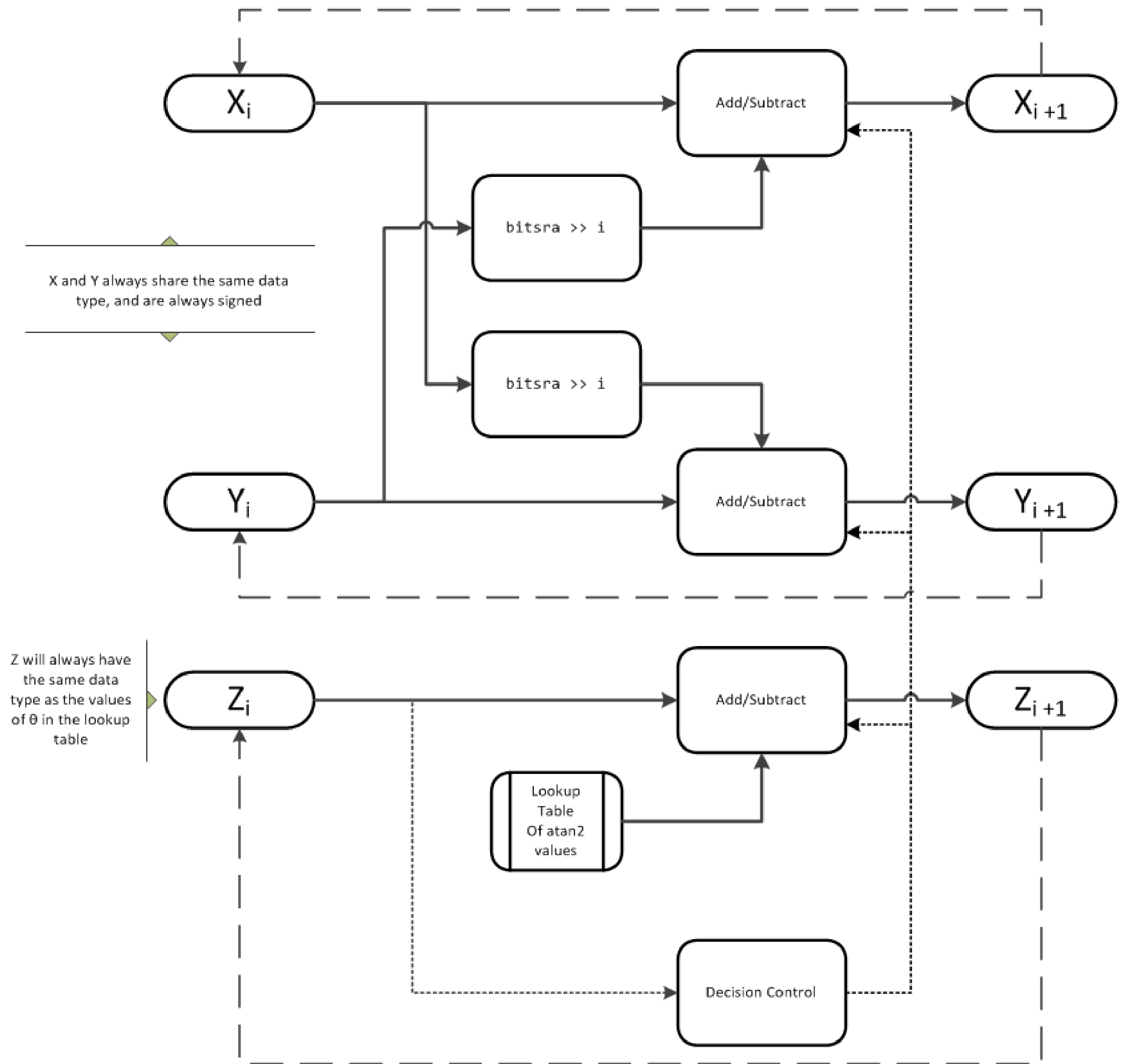
[4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Algorithms

Signal Flow Diagrams



CORDIC Rotation Kernel



X represents the real part, *Y* represents the imaginary part, and *Z* represents theta. This algorithm takes its initial values for *X*, *Y*, and *Z* from the inputs, `r` and `theta`.

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordicrotate` | `cordicsincos` | `pol2cart`

cordicrotate

Rotate input using CORDIC-based approximation

Syntax

```
v = cordicrotate(theta,u)
v = cordicrotate(theta,u,niters)
v = cordicrotate(theta,u,Name,Value)
v = cordicrotate(theta,u,niters,Name,Value)
```

Description

`v = cordicrotate(theta,u)` rotates the input `u` by `theta` using a CORDIC algorithm approximation. The function returns the result of `u` .* *e*^(*j**`theta`).

`v = cordicrotate(theta,u,niters)` performs `niters` iterations of the algorithm.

`v = cordicrotate(theta,u,Name,Value)` scales the output depending on the Boolean value, `b`.

`v = cordicrotate(theta,u,niters,Name,Value)` specifies both the number of iterations and the `Name,Value` pair for whether to scale the output.

Input Arguments

`theta`

`theta` can be a signed or unsigned scalar, vector, matrix, or *N*-dimensional array containing the angle values in radians. All values of `theta` must be in the range [–2π 2π).

`u`

`u` can be a signed or unsigned scalar value or have the same dimensions as `theta`. `u` can be real or complex valued.

`niters`

`niters` is the number of iterations the CORDIC algorithm performs. This argument is optional. When specified, `niters` must be a positive, integer-valued scalar. If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is the word length of `u` or one less than the word length of `theta`, whichever is smaller. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

`ScaleOutput`

`ScaleOutput` is a Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. This argument is optional. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Default: true

Output Arguments

v

v contains the approximated result of the CORDIC rotation algorithm. When the input u is floating point, the output v has the same data type as the input.

When the input u is a *signed* integer or fixed point data type, the output v is a signed fi object. This fi object has a word length that is two bits larger than that of u. Its fraction length is the same as the fraction length of u.

When the input u is an *unsigned* integer or fixed point, the output v is a signed fi object. This fi object has a word length that is three bits larger than that of u. Its fraction length is the same as the fraction length of u.

Examples

Run the following code, and evaluate the accuracy of the CORDIC-based complex rotation.

```
wrdLn = 16;
theta = fi(-pi/3, 1, wrdLn);
u     = fi(0.25 - 7.1i, 1, wrdLn);
uTeTh = double(u) .* exp(1i * double(theta));

fprintf('\n\nNITERS\tReal\t ERROR\t LSBs\t\tImag\tERROR\tLSBs\n');
fprintf('------\t-------\t ------\t ----\t\t-------\t------\t----\n');
for niters = 1:(wrdLn - 1)
 v_fi   = cordicrotate(theta, u, niters);
 v_dbl  = double(v_fi);
 x_err  = abs(real(v_dbl) - real(uTeTh));
  y_err  = abs(imag(v_dbl) - imag(uTeTh));
 fprintf('%d\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n',...
   niters, real(v_dbl),x_err,(x_err * pow2(v_fi.FractionLength)), ...
   imag(v_dbl),y_err, (y_err * pow2(v_fi.FractionLength)));
end
fprintf('\n');
```

The output table appears as follows:

```
NITERS  Real     ERROR    LSBs    Imag     ERROR    LSBs
------    -------     ------     ----        -------     ------     ------
1      -4.8438   1.1800   4833.5  -5.1973  1.4306  5859.8
2      -6.6567   0.6329   2592.5  -2.4824  1.2842  5260.2
3      -5.8560   0.1678   687.5   -4.0227  0.2560  1048.8
4      -6.3098   0.2860   1171.5  -3.2649  0.5018  2055.2
5      -6.0935   0.0697   285.5   -3.6528  0.1138  466.2
6      -5.9766   0.0472   193.5   -3.8413  0.0746  305.8
7      -6.0359   0.0121   49.5    -3.7476  0.0191  78.2
8      -6.0061   0.0177   72.5    -3.7947  0.0280  114.8
9      -6.0210   0.0028   11.5    -3.7710  0.0043  17.8
10     -6.0286   0.0048   19.5    -3.7590  0.0076  31.2
11     -6.0247   0.0009   3.5        -3.7651  0.0015  6.2
12     -6.0227   0.0011   4.5        -3.7683  0.0017  6.8
13     -6.0237   0.0001   0.5        -3.7666  0.0001  0.2
```

```
14     -6.0242   0.0004   1.5        -3.7656  0.0010  4.2
15     -6.0239   0.0001   0.5        -3.7661  0.0005  2.2
```

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

More About

[1] Volder, JE. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

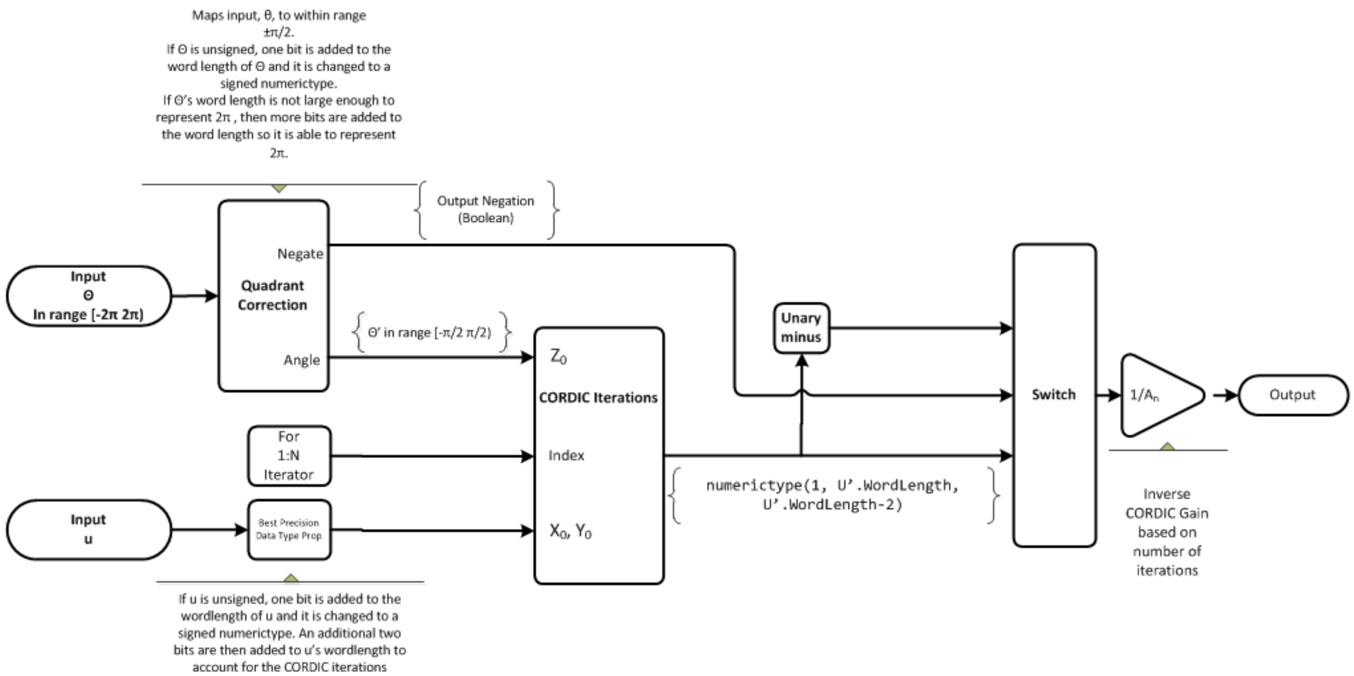
[2] Andraka, R. “A survey of CORDIC algorithm for FPGA based computers.” *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

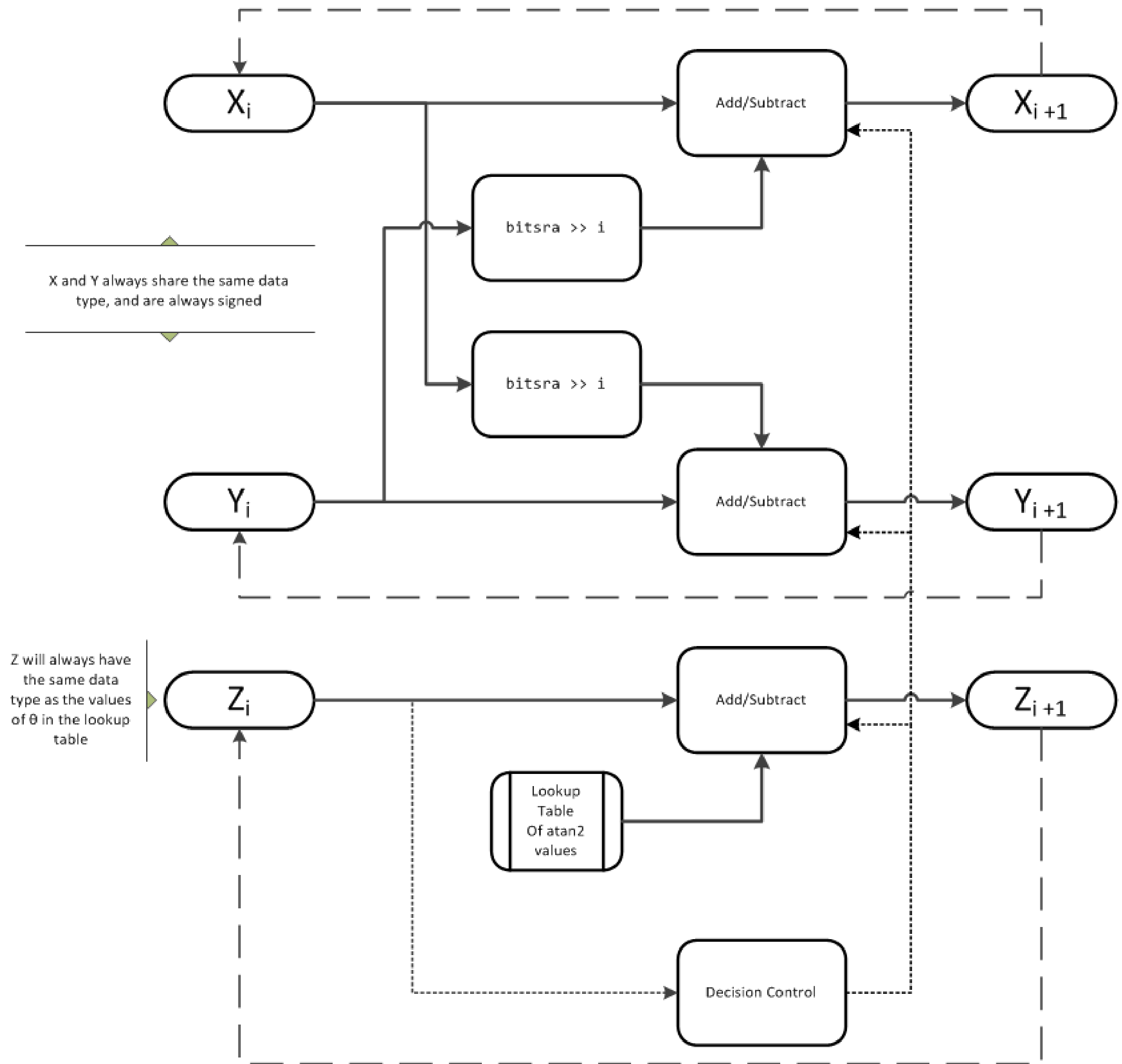
[4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Algorithms

Signal Flow Diagrams



CORDIC Rotation Kernel



X represents the real part, *Y* represents the imaginary part, and *Z* represents theta. This algorithm takes its initial values for *X*, *Y*, and *Z* from the inputs, `u` and `theta`.

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2011a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordicpol2cart` | `cordiccexp`

cordicsin

CORDIC-based approximation of sine

Syntax

```
y = cordicsin(theta)
y = cordicsin(theta,niters)
```

Description

`y = cordicsin(theta)` computes the sine of `theta` using a CORDIC algorithm approximation.

`y = cordicsin(theta,niters)` computes the sine of `theta` using a CORDIC algorithm approximation with specified number of iterations, `niters`.

Examples

Compare Results of `cordicsin` and `sin` Functions

This example compares the results produced by the `cordicsin` algorithm to the results of the double-precision `sin` function.

Create 1024 points between `[0, 2*pi)`.

```
stepSize = pi/512;
thRadDbl = 0:stepSize:(2*pi - stepSize);
thRadFxp = sfi(thRadDbl,12);        % signed, 12-bit fixed point
sinThRef = sin(double(thRadFxp));    % reference results
```

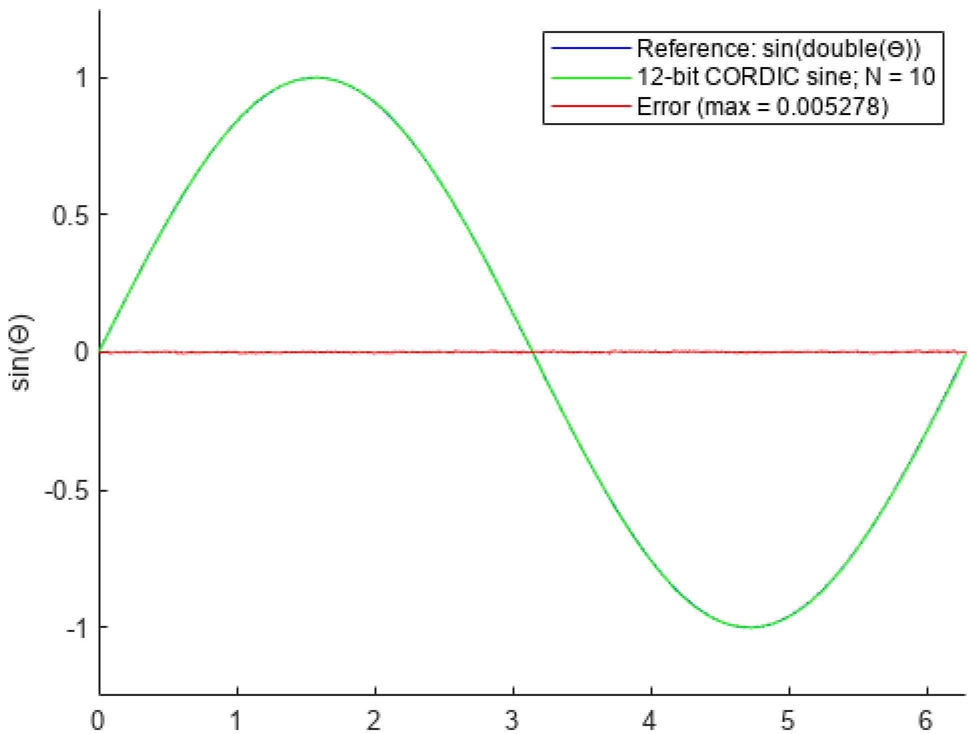
Set the number of iterations to 10.

```
niters = 10;
cdcSinTh  = cordicsin(thRadFxp,niters);
errCdcRef = sinThRef - double(cdcSinTh);
```

Compare the fixed-point `cordicsin` function results to the results of the double-precision `sin` function.

```
figure
hold on
axis([0 2*pi -1.25 1.25])
plot(thRadFxp,sinThRef,'b');
plot(thRadFxp,cdcSinTh,'g');
plot(thRadFxp,errCdcRef,'r');
ylabel('sin(\Theta)');
gca.XTick = 0:pi/2:2*pi;
gca.XTickLabel = {'0','pi/2','pi','3*pi/2','2*pi'};
gca.YTick = -1:0.5:1;
gca.YTickLabel = {'-1.0','-0.5','0','0.5','1.0'};
ref_str = 'Reference: sin(double(\Theta))';
cdc_str = sprintf('12-bit CORDIC sine; N = %d',niters);
```

```
err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
legend(ref_str,cdc_str,err_str);
```



After 10 iterations, the CORDIC algorithm has approximated the sine of theta to within 0.005492 of the double-precision sine result.

Input Arguments

`theta` — Input angle in radians

scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a signed or unsigned scalar, vector, matrix, or multidimensional array. All values of `theta` must be real and in the range [–2π 2π).

`niters` — Number of iterations

positive integer-valued scalar

Number of iterations the CORDIC algorithm performs, specified as a positive, integer-valued scalar.

If you do not specify `niters`, or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is 52 for double or 23 for single. Increasing the number of iterations can produce more accurate results, but it also increases the expense of the computation and adds latency.

Output Arguments

y — CORDIC-based approximation of sine
scalar | vector | matrix | multidimensional array

CORDIC-based approximation of sine of `theta`, returned as a scalar, vector, matrix, or multidimensional array.

When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength – 2`.

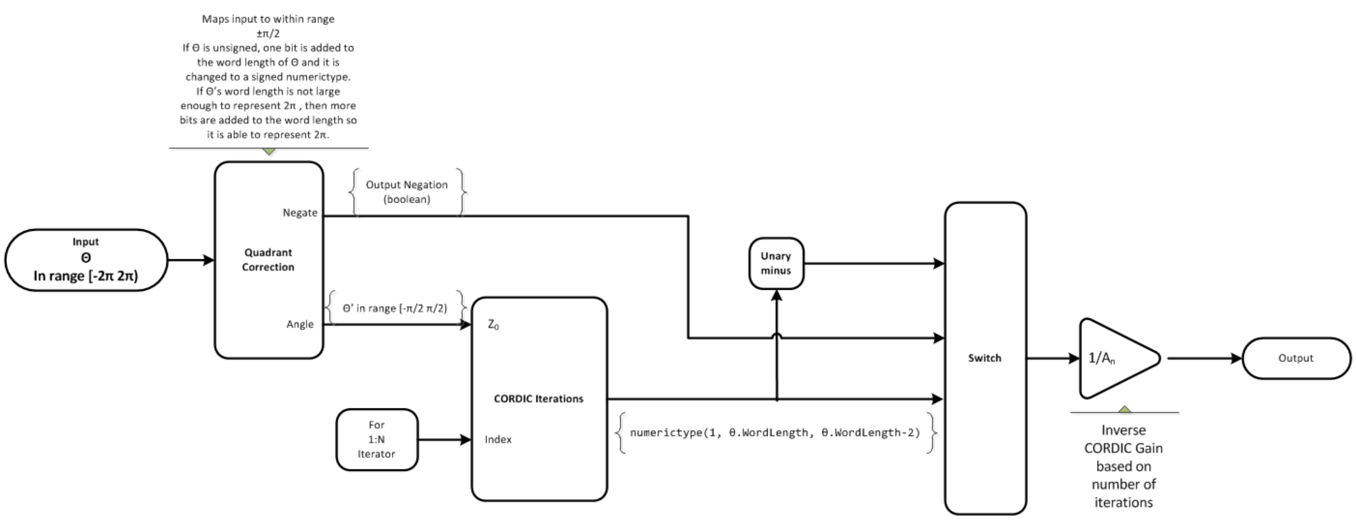
Algorithms

CORDIC

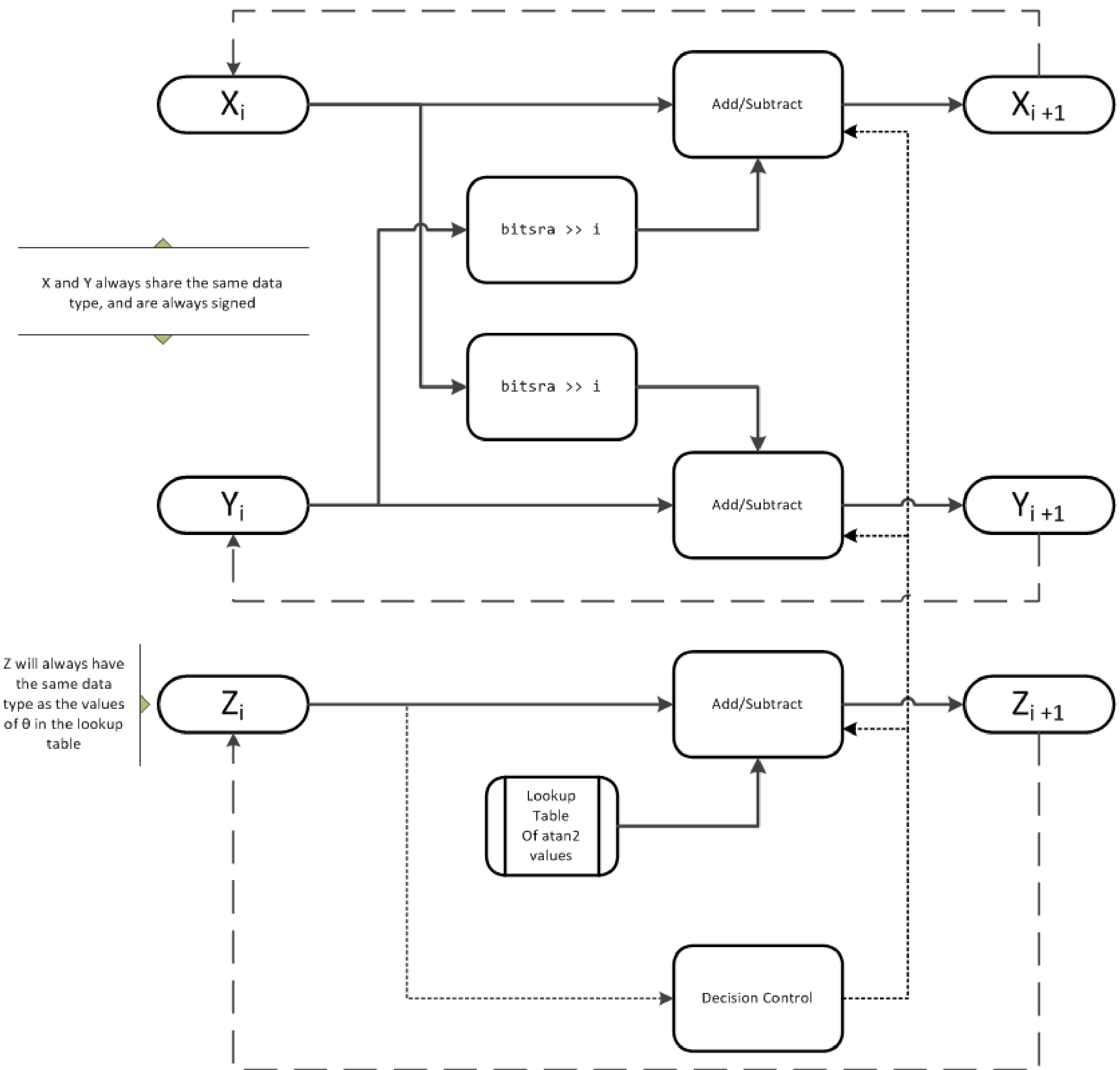
CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Signal Flow Diagrams



CORDIC Rotation Kernel



X represents the sine, *Y* represents the cosine, and *Z* represents theta. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses the following initial values:

z_0 is initialized to the θ input argument value

x_0 is initialized to $\frac{1}{A_N}$

y_0 is initialized to 0

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

You can generate HDL code for `cordicsin` function.

See Also

`cordiccexp` | `cordiccos` | `cordicsincos` | `sin` | `cos`

cordicsincos

CORDIC-based approximation of sine and cosine

Syntax

```
[y,x] = cordicsincos(theta)
[y,x] = cordicsincos(theta,niters)
```

Description

`[y,x] = cordicsincos(theta)` computes the sine and cosine of `theta` using a CORDIC algorithm approximation. `y` contains the approximated sine result, and `x` contains the approximated cosine result.

`[y,x] = cordicsincos(theta,niters)` performs the number of CORDIC algorithm iterations specified by `niters`.

Examples

Compare Accuracy of CORDIC Sine and Cosine Results

View the effect of the number of iterations on the result of the CORDIC approximation of sine and cosine.

```
wordlength = 8;
theta = fi(pi/2,1,wordlength);

fprintf('\n\nNITERS\t\tY (SIN)\t ERROR\t LSBs\t\tX (COS)\t ERROR\t LSBs\n');
fprintf('------\t\t-------\t ------\t ----\t\t-------\t ------\t ----\n');

for niters = 1:(wordlength - 1)
  [y,x] = cordicsincos(theta,niters);
  y_FL   = y.FractionLength;
  y_dbl  = double(y);
  x_dbl  = double(x);
  y_err  = abs(y_dbl - sin(double(theta)));
  x_err  = abs(x_dbl - cos(double(theta)));
  fprintf(' %d\t\t%1.4f\t %1.4f\t %1.1f\t\t%1.4f\t %1.4f\t %1.1f\n', ...
      niters,y_dbl,y_err,(y_err*pow2(y_FL)),x_dbl,x_err, ...
      (x_err*pow2(y_FL)));
end

fprintf('\n');
```

```
NITERS        Y (SIN)     ERROR     LSBs        X (COS)     ERROR     LSBs
------        -------     ------    ----        -------     ------    ----
 1         0.7031     0.2968     19.0        -0.7031     0.6958     44.5
 2         0.9375     0.0625     4.0        -0.3125     0.3052     19.5
 3         0.9688     0.0312     2.0        -0.0625     0.0552     3.5
```

```
4        0.9688     0.0312     2.0        0.0625     0.0698     4.5
5        0.9844     0.0156     1.0        0.0000     0.0073     0.5
6        0.9844     0.0156     1.0        0.0312     0.0386     2.5
7        1.0000     0.0000     0.0        0.0156     0.0230     1.5
```

Input Arguments

`theta` — Input angle in radians

scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array. All `theta` values must be in the range `[-2*pi,2*pi)`. When `theta` has a fixed-point data type, it must be signed.

`niters` — Number of iterations CORDIC algorithm performs

positive integer-valued scalar

Number of iterations the CORDIC algorithm performs, specified as a positive integer-valued scalar.

If you do not specify `niters` or if you specify a value that is too large, the algorithm uses a maximum value. For fixed-point operation, the maximum number of iterations is one less than the word length of `theta`. For floating-point operation, the maximum value is `52` for double or `23` for single.

Increasing the number of iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Output Arguments

y — CORDIC-based approximated sine of `theta`

scalar | vector | matrix | multidimensional array

CORDIC-based approximated sine of `theta`, returned as a scalar, vector, matrix, or multidimensional array. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input and a fraction length equal to the `Wordlength - 2`.

x — CORDIC-based approximated cosine of `theta`

scalar | vector | matrix | multidimensional array

CORDIC-based approximated cosine of `theta`, returned as a scalar, vector, matrix, or multidimensional array. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input and a fraction length equal to the `Wordlength - 2`.

More About

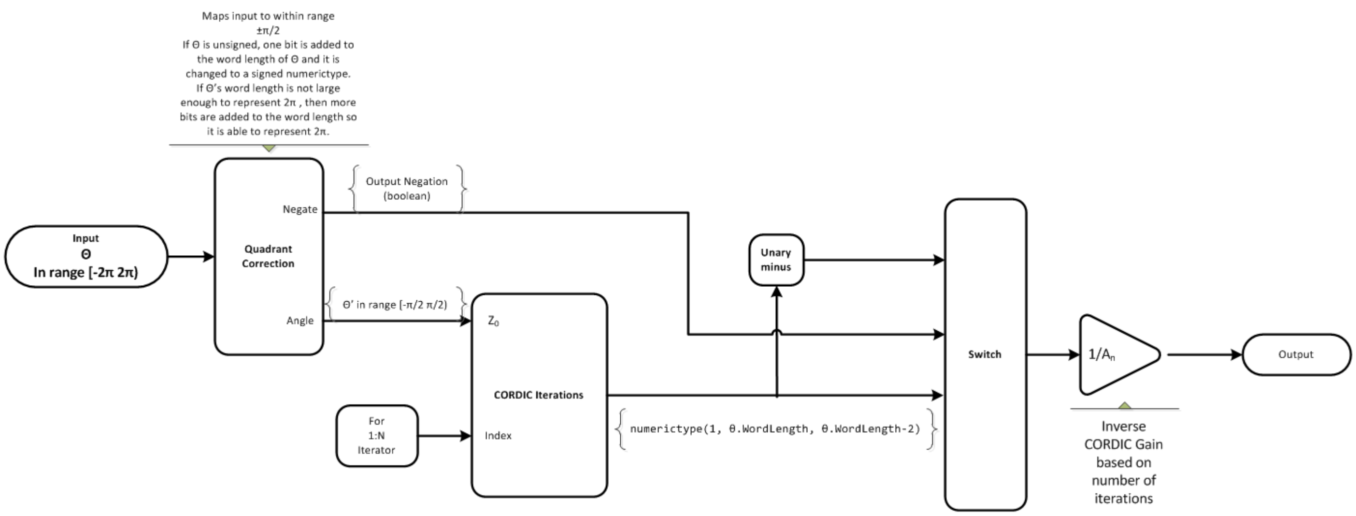
CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see “References” on page 4-316). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arcsine, arccosine, arctangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

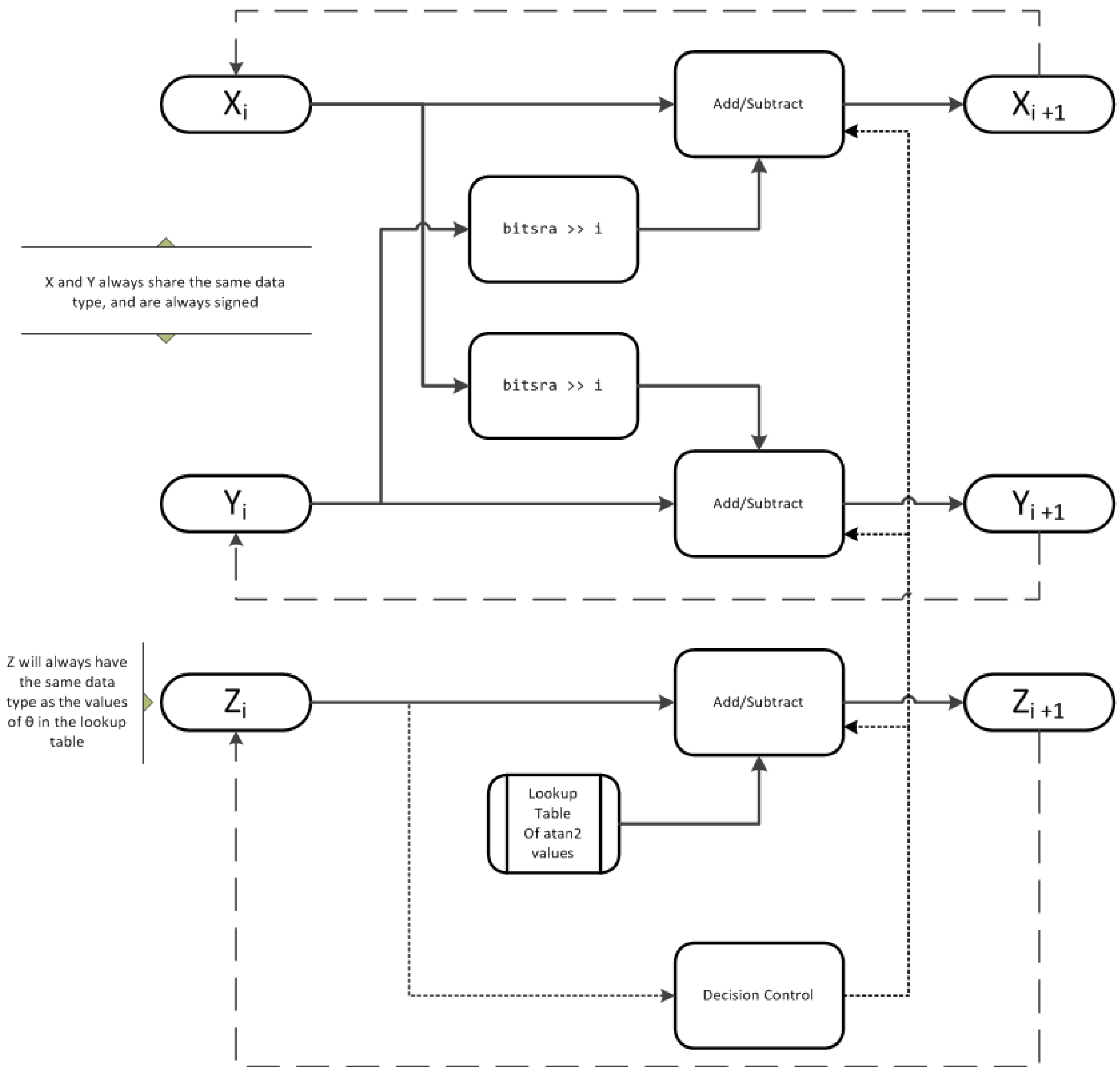
Increasing the number of CORDIC iterations can produce more accurate results but also increases the expense of the computation and adds latency.

Algorithms

Signal Flow Diagrams



CORDIC Rotation Kernel



X represents the sine, *Y* represents the cosine, and *Z* represents *theta*. The accuracy of the CORDIC rotation kernel depends on the choice of initial values for *X*, *Y*, and *Z*. This algorithm uses these initial values:

z_0 is initialized to the θ input argument value

x_0 is initialized to $\frac{1}{A_N}$

y_0 is initialized to 0

`fimath` Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction` — `Wrap`
- `RoundingMethod` — `Floor`

The output has no attached `fimath`.

Version History

Introduced in R2010a

References

[1] Volder, Jack E. “The CORDIC Trigonometric Computing Technique.” *IRE Transactions on Electronic Computers*. EC-8, no. 3 (Sept. 1959): 330–334.

[2] Andraka, Ray. “A Survey of CORDIC Algorithm for FPGA Based Computers.” In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 191-200. https://dl.acm.org/doi/10.1145/275107.275139.

[3] Walther, J.S. “A Unified Algorithm for Elementary Functions.” In *Proceedings of the May 18-20, 1971 Spring Joint Computer Conference*, 379-386. https://dl.acm.org/doi/10.1145/1478786.1478840.

[4] Schelin, Charles W. “Calculator Function Approximation.” *The American Mathematical Monthly*, no. 5 (May 1983): 317-325. https://doi.org/10.2307/2975781.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`cordiccexp` | `cordiccos` | `cordicsin`

cordicsqrt

CORDIC-based approximation of square root

Syntax

```
y=cordicsqrt(u)
y=cordicsqrt(u, niters)
y=cordicsqrt( ___ , 'ScaleOutput', B)
```

Description

`y=cordicsqrt(u)` computes the square root of `u` using a CORDIC algorithm implementation.

`y=cordicsqrt(u, niters)` computes the square root of `u` by performing `niters` iterations of the CORDIC algorithm.

`y=cordicsqrt( ___ , 'ScaleOutput', B)` scales the output depending on the Boolean value of `B`.

Examples

Calculate the CORDIC Square Root

Find the square root of `fi` object `x` using a CORDIC implementation.

```
x = fi(1.6,1,12);
y = cordicsqrt(x)
```

```
y = 
    1.2646

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 10
```

Because you did not specify `niters`, the function performs the maximum number of iterations, `x.WordLength - 1`.

Compute the difference between the results of the `cordicsqrt` function and the double-precision `sqrt` function.

```
err = abs(sqrt(double(x))-double(y))
```

```
err = 1.0821e-04
```

Calculate the CORDIC Square Root With a Specified Number of Iterations

Compute the square root of x with three iterations of the CORDIC kernel.

```
x = fi(1.6,1,12);
y = cordicsqrt(x,3)
```

```
y = 
    1.2646

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 10
```

Compute the difference between the results of the `cordicsqrt` function and the double-precision `sqrt` function.

```
err = abs(sqrt(double(x))-double(y))
```

```
err = 1.0821e-04
```

Calculate the CORDIC Square Root Without Scaling the Output

```
x = fi(1.6,1,12);
y = cordicsqrt(x, 'ScaleOutput', 0)
```

```
y = 
    1.0479

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 10
```

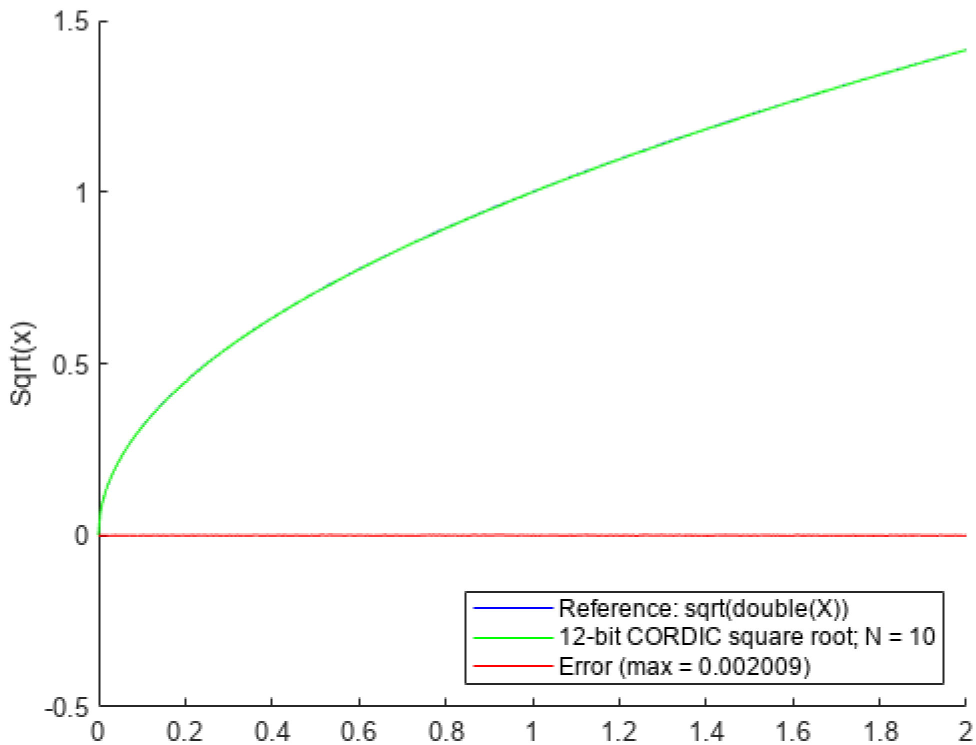
The output, `y`, was not scaled by the inverse CORDIC gain factor.

Compare Results of cordicsqrt and sqrt Functions

Compare the results produced by 10 iterations of the `cordicsqrt` algorithm to the results of the double-precision `sqrt` function.

```
% Create 500 points between [0, 2)
stepSize = 2/500;
XDbl = 0:stepSize:2;
XFxp = fi(XDbl, 1, 12);    % signed, 12-bit fixed-point
sqrtXRef = sqrt(double(XFxp));   % reference results

% Use 12-bit quantized inputs and set the number
% of iterations to 10.
% Compare the  fixed-point CORDIC results to the
% double-precision sqrt function results.

niters = 10;
cdcSqrtX  = cordicsqrt(XFxp,  niters);
errCdcRef = sqrtXRef - double(cdcSqrtX);
figure
```

```
hold on
axis([0 2 -.5 1.5])
plot(XFxp, sqrtXRef,  'b')
plot(XFxp, cdcSqrtX,  'g')
plot(XFxp, errCdcRef, 'r')
ylabel('Sqrt(x)')
gca.XTick = 0:0.25:2;
gca.XTickLabel = {'0','0.25','0.5','0.75','1','1.25','1.5','1.75','2'};
gca.YTick = -.5:.25:1.5;
gca.YTickLabel = {'-0.5','-0.25','0','0.25','0.5','0.75','1','1.25','1.5'};
ref_str = 'Reference: sqrt(double(X))';
cdc_str = sprintf('12-bit CORDIC square root; N = %d', niters);
err_str = sprintf('Error (max = %f)', max(abs(errCdcRef)));
legend(ref_str, cdc_str, err_str, 'Location', 'southeast')
```



Input Arguments

u — Data input array

scalar | vector | matrix | multidimensional array

Data input array, specified as a positive scalar, vector, matrix, or multidimensional array of fixed-point or built-in data types. When the input array contains values between 0.5 and 2, the algorithm is most accurate. A pre- and post-normalization process is performed on input values outside of this range. For more information on this process, see “Pre- and Post-Normalization” on page 4-322.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`niters` — Number of iterations
scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is `u.WordLength - 1`. For floating-point inputs, the default value of `niters` is 52 for double precision; 23 for single precision.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ScaleOutput` — Whether to scale the output
true (default) | false

Boolean value that specifies whether to scale the output by the inverse CORDIC gain factor. If you set `ScaleOutput` to `true` or `1`, the output values are multiplied by a constant, which incurs extra computations. If you set `ScaleOutput` to `false` or `0`, the output is not scaled.

Data Types: `logical`

Output Arguments

y — Output array
scalar | vector | matrix | multidimensional array

Output array, returned as a scalar, vector, matrix, or multidimensional array.

More About

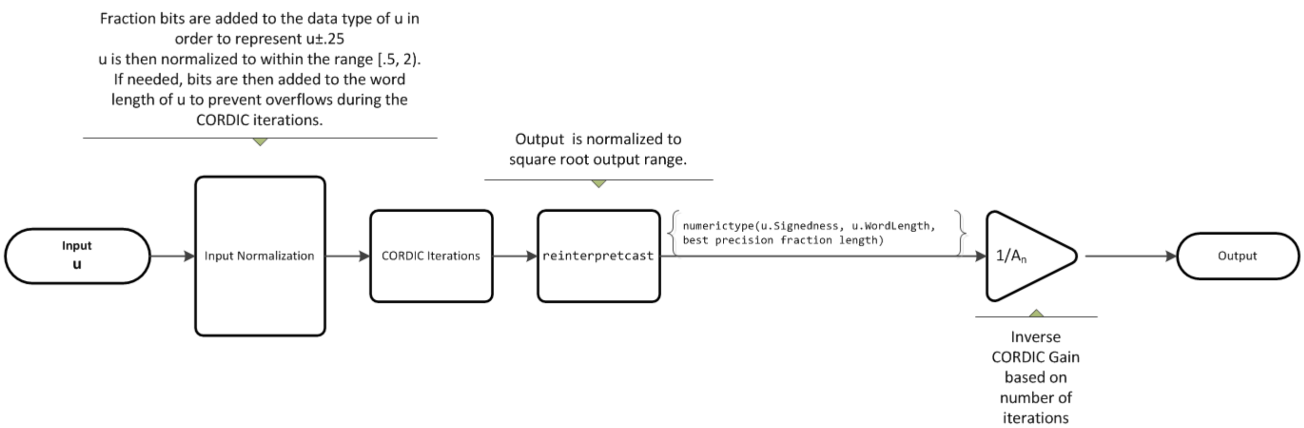
CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

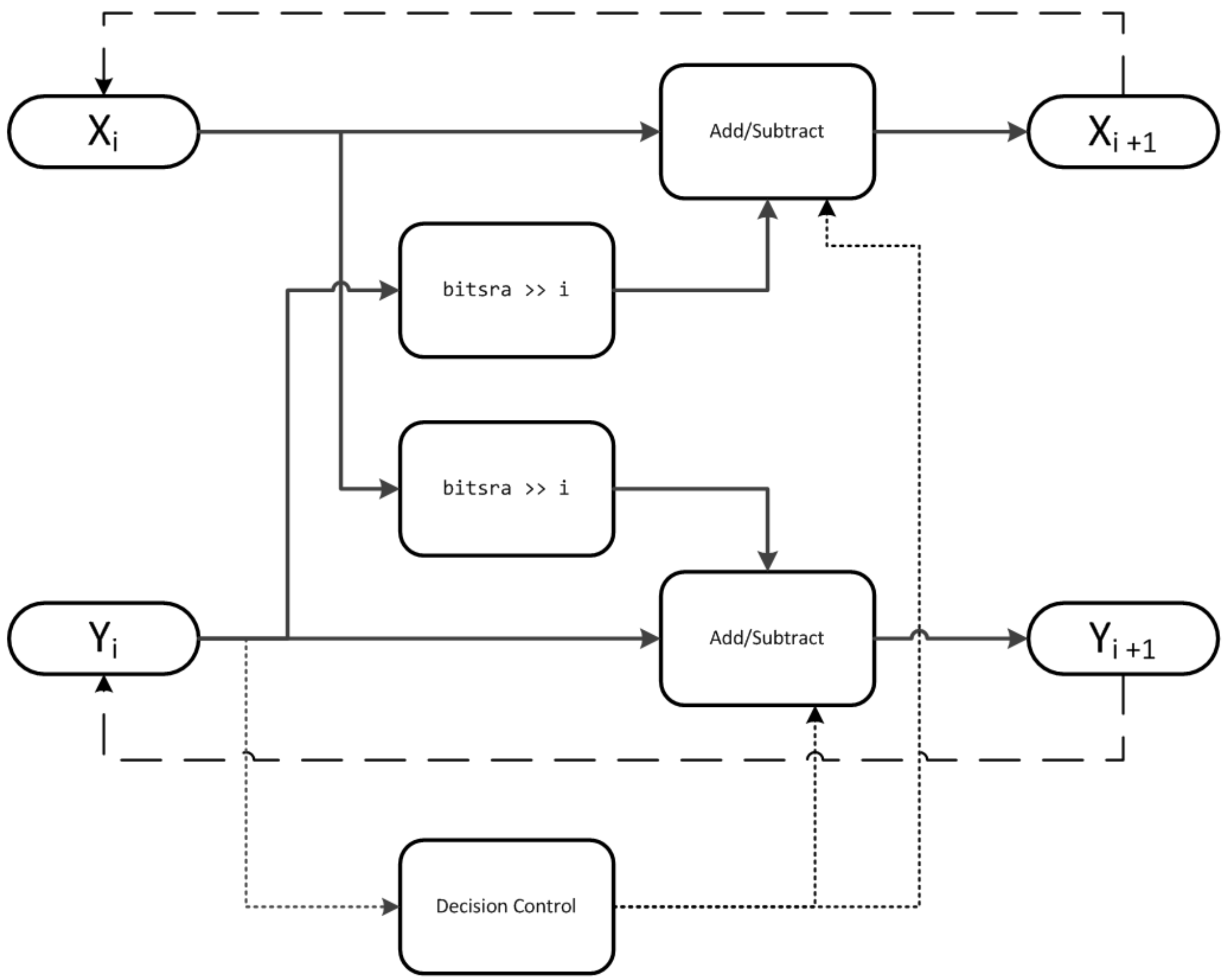
Algorithms

Signal Flow Diagrams



For further details on the pre- and post-normalization process, see “Pre- and Post-Normalization” on page 4-322.

CORDIC Hyperbolic Kernel



X is initialized to `u'+.25`, and *Y* is initialized to `u'-.25`, where `u'` is the normalized function input.

With repeated iterations of the CORDIC hyperbolic kernel, *X* approaches $A_N\sqrt{u'}$, where A_N represents the CORDIC gain. *Y* approaches `0`.

Pre- and Post-Normalization

For input values outside of the range of [0.5, 2) a pre- and post-normalization process occurs. This process performs bitshifts on the input array before passing it to the CORDIC kernel. The result is then shifted back into the correct output range during the post-normalization stage. For more details on this process see "Overcoming Algorithm Input Range Limitations" in "Compute Square Root Using CORDIC".

fimath Propagation Rules

CORDIC functions discard any local `fimath` attached to the input.

The CORDIC functions use their own internal `fimath` when performing calculations:

- `OverflowAction`—`Wrap`
- `RoundingMethod`—`Floor`

The output has no attached `fimath`.

Version History

Introduced in R2014a

References

[1] Volder, JE. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers*. Vol. EC-8, September 1959, pp. 330–334.

[2] Andraka, R. "A survey of CORDIC algorithm for FPGA based computers." *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. Feb. 22–24, 1998, pp. 191–200.

[3] Walther, J.S. "A Unified Algorithm for Elementary Functions." Hewlett-Packard Company, Palo Alto. Spring Joint Computer Conference, 1971, pp. 379–386. (from the collection of the Computer History Museum). www.computer.org/csdl/proceedings/afips/1971/5077/00/50770379.pdf

[4] Schelin, Charles W. "Calculator Function Approximation." *The American Mathematical Monthly*. Vol. 90, No. 5, May 1983, pp. 317–325.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-size signals are not supported.
- The number of iterations the CORDIC algorithm performs, `niters`, must be a constant.

See Also

`sqrt`

Topics
"Compute Square Root Using CORDIC"

cordictanh

CORDIC-based hyperbolic tangent

Syntax

```
T = cordictanh(theta)
T = cordictanh(theta, niters)
```

Description

`T = cordictanh(theta)` returns the hyperbolic tangent of `theta`.

`T = cordictanh(theta, niters)` returns the hyperbolic tangent of theta by performing `niters` iterations of the CORDIC algorithm.

Examples

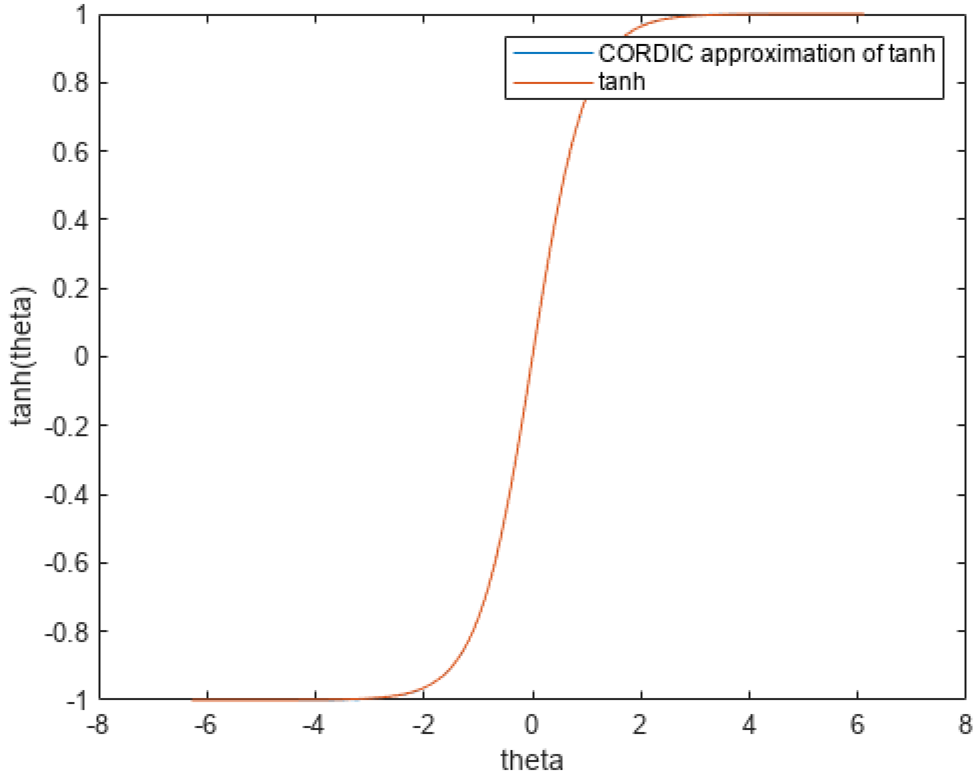
Compute CORDIC Hyperbolic Tangent

Find the hyperbolic tangent of `fi` object `theta` using a CORDIC implementation with the default number of iterations.

```
theta = fi(-2*pi:.1:2*pi-.1);
T_cordic = cordictanh(theta);
```

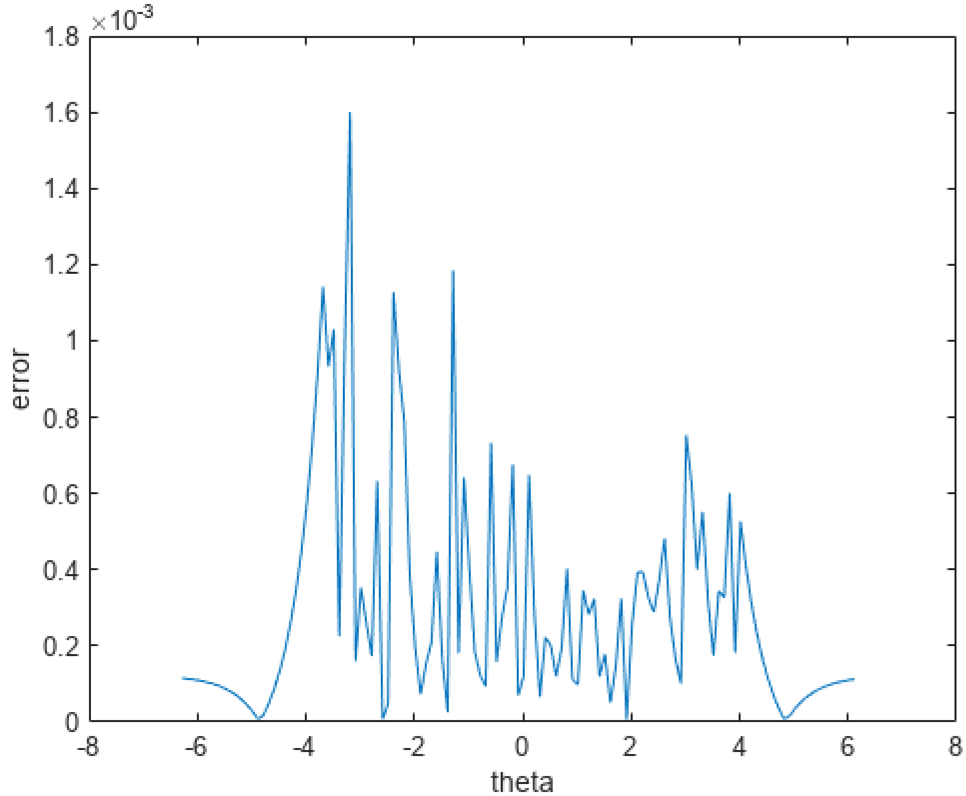
Plot the hyperbolic tangent of `theta` using the `tanh` function and its CORDIC approximation.

```
T = tanh(double(theta));
plot(theta, T_cordic);
hold on;
plot(theta, T);
legend('CORDIC approximation of tanh', 'tanh');
xlabel('theta');
ylabel('tanh(theta)');
```



Compute the difference between the results of the `cordictanh` function and the `tanh` function.

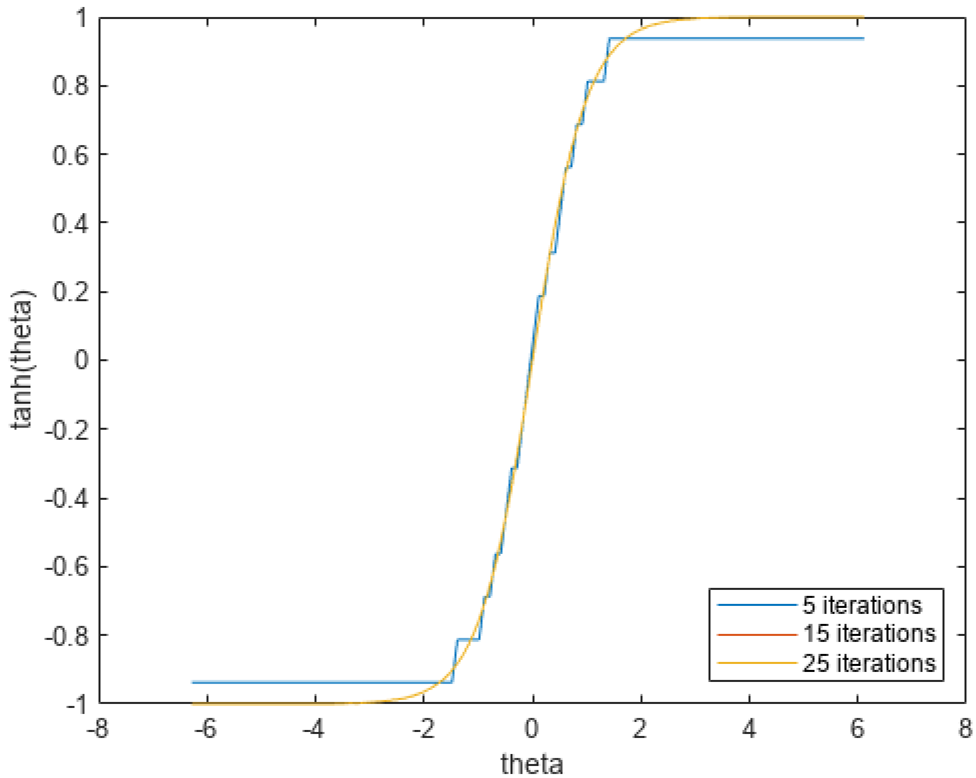
```
figure;
err = abs(T - double(T_cordic));
plot(theta, err);
xlabel('theta');
ylabel('error');
```



Compute CORDIC Hyperbolic Tangent with Specified Number of Iterations

Find the hyperbolic tangent of `fi` object `theta` using a CORDIC implementation and specify the number of iterations the CORDIC kernel should perform. Plot the CORDIC approximation of the hyperbolic tangent of theta with varying numbers of iterations.

```
theta = fi(-2*pi:.1:2*pi-.1);
for niters = 5:10:25
T_cordic = cordictanh(theta,niters);
plot(theta,T_cordic);
hold on;
end
xlabel('theta');
ylabel('tanh(theta)');
legend('5 iterations','15 iterations',...
    '25 iterations','Location','southeast');
```



Input Arguments

theta — angle values
scalar | vector | matrix | n-dimensional array

Angle values in radians specified as a scalar, vector, matrix, or N-dimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

niters — Number of iterations
scalar

The number of iterations that the CORDIC algorithm performs, specified as a positive, integer-valued scalar. If you do not specify `niters`, the algorithm uses a default value. For fixed-point inputs, the default value of `niters` is one less than the word length of the input array, `theta`. For double-precision inputs, the default value of `niters` is 52. For single-precision inputs, the default value is 23.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

T — Output array
scalar | vector | matrix | n-dimensional array

`T` is the CORDIC-based approximation of the hyperbolic tangent of `theta`. When the input to the function is floating point, the output data type is the same as the input data type. When the input is fixed point, the output has the same word length as the input, and a fraction length equal to the `WordLength` – 2.

Version History

Introduced in R2017b

See Also

`cordicatan2` | `cordicsin` | `cordiccos` | `tanh`

cos

Package: `embedded`

Cosine of `fi` object in radians

Syntax

`Y = cos(X)`

Description

`Y = cos(X)` returns the cosine for each element of `fi` input `X` using an 8-bit lookup table algorithm.

Examples

Cosine of Fixed-Point Angles

Calculate the cosine of fixed-point input values.

```
X = fi([0,pi/4,pi/3,pi/2,(2*pi)/3,(3*pi)/4,pi])
```

```
X =
         0    0.7854    1.0472    1.5708    2.0944    2.3562    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
Y = cos(X)
```

```
Y =
    1.0000    0.7072    0.4999    0.0001   -0.4999   -0.7070   -1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Input Arguments

X — Input angle in radians
scalar | vector | matrix | multidimensional array

Input angle in radians, specified as a scalar, vector, matrix, or multidimensional array.

`X` can be a real-valued, signed or unsigned:

- `fi` single

- `fi` double
- `fi` fixed-point with binary-point scaling
- `fi` scaled double with binary-point scaling

Example: `X = fi([pi pi/6],1,8);`

Data Types: `fi`

Output Arguments

Y — Cosine of input angle
scalar | vector | matrix | multidimensional array

Cosine of input angle, returned as a real-valued `fi` scalar, vector, matrix, or multidimensional array.

More About

Cosine

The cosine of angle Θ is defined as

$$\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2}$$

Algorithms

The `cos` function computes the cosine of fixed-point input using an 8-bit lookup table as follows:

1. Perform a modulo 2π, so the input is in the range [0,2π) radians.
2. Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
3. Compute the table index, based on the 16-bit stored integer value, normalized to the full `uint16` range.
4. Use the 8 most-significant bits to obtain the first value from the table.
5. Use the next-greater table value as the second value.
6. Use the 8 least-significant bits to interpolate between the first and second values, using nearest-neighbor linear interpolation.

`fimath` Propagation Rules

The `cos` function ignores and discards any `fimath` attached to the input, X. The output, Y, is always associated with the default `fimath`.

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`cos` | `angle` | `sin` | `atan2` | `cordiccos` | `cordicsin`

ctranspose

Complex conjugate transpose of `fi` object

Syntax

```
ctranspose(a)
```

Description

This function accepts `fi` objects as inputs.

`ctranspose(a)` returns the complex conjugate transpose of `fi` object `a`. It is also called for the syntax `a'`.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

CustomFloat

Numeric object with a custom floating-point data type

Description

Use a `CustomFloat` object to define a floating-point numeric data type with specified word length and mantissa length. Floating-point data types defined by a `CustomFloat` object adhere to the IEEE 754-2008 standard. For more information on floating-point data types, see “Floating-Point Numbers”.

Creation

Syntax

```
x = CustomFloat(v)
x = CustomFloat(v, type)
x = CustomFloat(v, WordLength, MantissaLength)
x = CustomFloat(v, WordLength, MantissaLength, 'typecast')
x = CustomFloat(cf)
```

Description

`x = CustomFloat(v)` returns a `CustomFloat` object with value `v`. The output object has the same word length, mantissa length, and exponent length as input `v`.

`x = CustomFloat(v, type)` returns a `CustomFloat` object with value `v` and floating-point type specified by `type`.

`x = CustomFloat(v, WordLength, MantissaLength)` returns a `CustomFloat` object with the specified word length and mantissa length.

`x = CustomFloat(v, WordLength, MantissaLength, 'typecast')` returns a `CustomFloat` object with the bit pattern of `v` and the specified mantissa length. The word length must match the word length of the input `v`.

`x = CustomFloat(cf)` returns a `CustomFloat` object with value and data type properties of `CustomFloat` object `cf`.

Input Arguments

v — Value of object
scalar | vector | matrix | multi-dimensional array

The value of the `CustomFloat` object, specified as a scalar, vector, matrix, or multi-dimensional array.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

type — Floating-point type of object
'double' | 'single' | 'half'

Floating-point data type of CustomFloat object, specified as either 'double', 'single', or 'half'.

The properties of these types are summarized in the following table.

Type	Word Length	Mantissa Length
double	64	52
single	32	23
half	16	10

Data Types: char

cf — Custom floating-point type
CustomFloat object

Custom floating-point type, specified as a CustomFloat object.

Properties

ExponentBias — Offset value for the exponent
scalar integer

Scalar integer representing the offset value for the exponent.

This property cannot be changed directly, however you can change this property by changing the WordLength and MantissaLength properties, which influence the ExponentLength property. The ExponentBias for a floating-point data type is computed through the following equation:

$$ExponentBias = 2^{e\text{-}1}\text{-}1 \quad (4\text{-}6)$$

where e represents the ExponentLength.

Data Types: double

ExponentLength — Number of bits representing the exponent
scalar integer less than 31

Number of bits representing the exponent. You cannot edit this property directly, however you can change the exponent length by changing the MantissaLength and WordLength properties.

The ExponentLength, MantissaLength, and WordLength properties are related through the following equation:

$$WordLength = 1 + MantissaLength + ExponentLength \quad (4\text{-}7)$$

ExponentLength must be less than 31 bits.

Data Types: double

MantissaLength — Number of bits representing the mantissa
scalar integer

Number of bits representing the mantissa, specified as a scalar integer.

The `ExponentLength`, `MantissaLength`, and `WordLength` properties are related through the following equation.

$$WordLength = 1 + MantissaLength + ExponentLength \tag{4-8}$$

Note `ExponentLength` must be less than 31 bits.

Example: `custfloat.MantissaLength = 14;`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`WordLength` — Total number of bits in the data type
scalar integer

Total number of bits in the data type, specified as a scalar integer.

The `ExponentLength`, `MantissaLength`, and `WordLength` properties are related through the following equation.

$$WordLength = 1 + MantissaLength + ExponentLength \tag{4-9}$$

Note `ExponentLength` must be less than 31 bits.

Example: `custfloat.WordLength = 28;`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Object Functions

Math and Arithmetic

abs	Absolute value and complex magnitude
ceil	Round toward positive infinity
complex	Create complex array
conj	Complex conjugate
cosh	Hyperbolic cosine
exp	Exponential
fix	Round toward zero
floor	Round toward negative infinity
fma	Multiply and add using fused multiply add approach
hypot	Square root of sum of squares (hypotenuse)
ldivide	Left array division
log	Natural logarithm
log2	Base 2 logarithm and floating-point number dissection
log10	Common logarithm (base 10)
minus	Subtraction
mod	Remainder after division (modulo operation)

mtimes	Matrix multiplication
ndims	Number of array dimensions
plus	Add numbers, append strings
pow10	Base 10 power and scale half-precision numbers
pow2	Base 2 exponentiation and scaling of floating-point numbers
power	Element-wise power
rdivide	Right array division
real	Real part of complex number
rem	Remainder after division
round	Round to nearest decimal or integer
rsqrt	Reciprocal square root
sqrt	Square root
tanh	Hyperbolic tangent
times	Multiplication
uminus	Unary minus
uplus	Unary plus

Data Types

bin	Unsigned binary representation of stored integer of fi object
double	Double-precision arrays
fi	Construct fixed-point numeric object
int8	8-bit signed integer arrays
int16	16-bit signed integer arrays
int32	32-bit signed integer arrays
int64	64-bit signed integer arrays
isnan	Determine which array elements are NaN
isreal	Determine whether array uses complex storage
single	Single-precision arrays
uint8	8-bit unsigned integer arrays
uint16	16-bit unsigned integer arrays
uint32	32-bit unsigned integer arrays
uint64	64-bit unsigned integer arrays

Relational and Logical Operators

eq	Determine equality
ge	Determine greater than or equal to
gt	Determine greater than
le	Determine less than or equal to
lt	Determine less than
ne	Determine inequality

Array and Matrix Operations

cat	Concatenate arrays
ctranspose	Complex conjugate transpose
horzcat	Concatenate arrays horizontally
isfinite	Determine which array elements are finite
isinf	Determine which array elements are infinite
norm	Vector and matrix norms
numel	Number of array elements
reshape	Reshape array

size	Array size
transpose	Transpose vector or matrix
vertcat	Concatenate arrays vertically

Language Fundamentals

disp	Display value of variable

Examples

Create a CustomFloat Object

This example shows how to create a `CustomFloat` object.

```
v = pi;
x = CustomFloat(v)
```

```
x = 
    3.1416


           Data Type: Floating-point: Double-precision
          WordLength:  64
      MantissaLength:  52
      ExponentLength:  11
        ExponentBias: 1023
```

Because the input to the `CustomFloat` constructor was a double, the data type of the `CustomFloat` object, `x`, is also a double. If the value passed in to the `CustomFloat` function is a single, then the resulting `CustomFloat` object will also have a single-precision floating-point data type.

```
v = single(pi);
x = CustomFloat(v)
```

```
x = 
    3.1416


           Data Type: Floating-point: Single-precision
          WordLength:  32
      MantissaLength:  23
      ExponentLength:   8
        ExponentBias: 127
```

Create a Half-Precision CustomFloat Object

To create a `CustomFloat` object with a specified floating-point data type, specify the data type as the second argument in the `CustomFloat` function.

```
v = pi;
x = CustomFloat(v,'half')
```

```
x = 
    3.1406
```

```
         Data Type: Floating-point: Half-precision
        WordLength:  16
    MantissaLength:  10
    ExponentLength:   5
      ExponentBias:  15
```

Create a CustomFloat Object with Specified Word Length and Mantissa Length

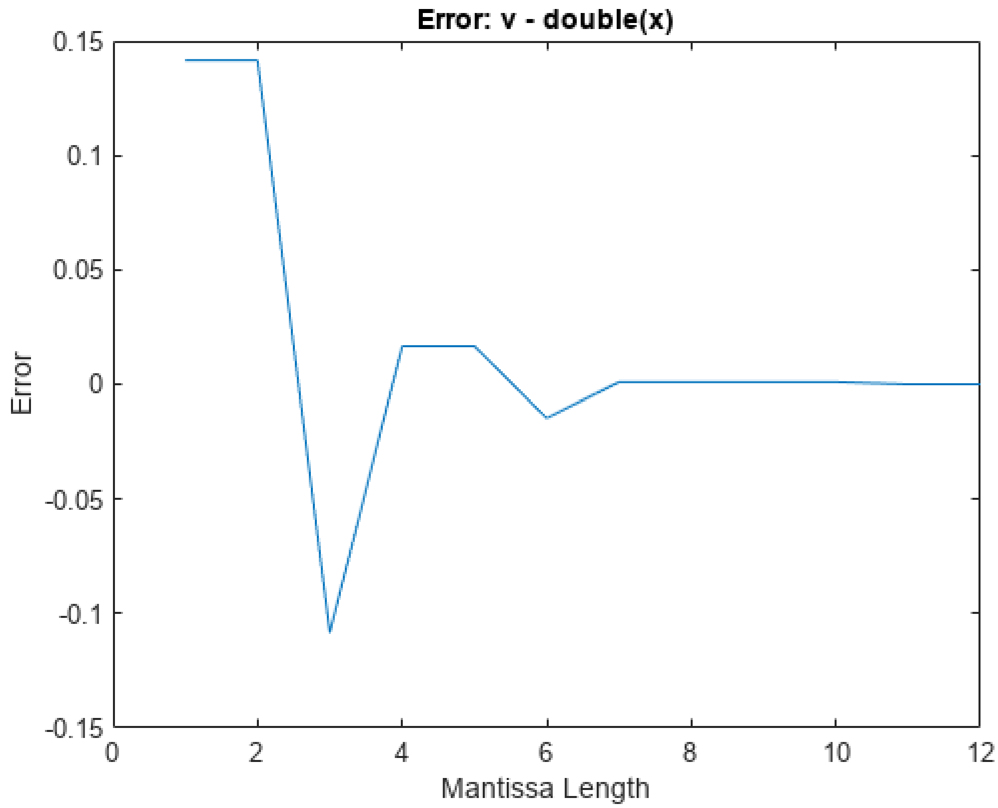
Specify a word length and a mantissa length in the `CustomFloat` function.

```
v = pi;
wl = 16;
ml = 4;
x = CustomFloat(v,wl,ml)
```

```
x = 
    3.1250


           Data Type: Floating-point: Custom-precision
          WordLength:  16
      MantissaLength:   4
      ExponentLength:  11
        ExponentBias: 1023
```

Compare the difference between the double-precision value and the value of the `CustomFloat` object as you change the mantissa length.

```
err = zeros(1,12);
for ml = 1:12
    x = CustomFloat(v,wl,ml);
    err(ml) = v-double(x);
end

plot(err);
title('Error: v - double(x)');
ylabel('Error');
xlabel('Mantissa Length');
```



Typecast a Value to a New CustomFloat Data Type

Using the `'typecast'` input argument, the `CustomFloat` function creates a `CustomFloat` object with the bit pattern of the input value, and the specified word length and mantissa length.

Define a single-precision value. Single-precision floating-point data types have a 32-bit word length and 23-bit mantissa length. View the binary representation of the single-precision value.

```
v = single(pi);
bit_pattern = bin(CustomFloat(v))
```

```
bit_pattern =
'01000000010010010000111111011011'
```

Define a `CustomFloat` object that has the same bit pattern as the input value, but has a different mantissa length.

```
x = CustomFloat(v, 32, 20, 'typecast')
```

```
x =
   50.1239


           Data Type: Floating-point: Custom-precision
          WordLength:  32
```

```
      MantissaLength:  20
      ExponentLength:  11
        ExponentBias: 1023
```

View the binary representation of the `CustomFloat` object, and compare it to the bit pattern of the single-precision input value.

```
bit_pattern2 = bin(x)
```

```
bit_pattern2 =
'01000000010010010000111111011011'
```

```
same = strcmp(bit_pattern, bit_pattern2)
```

```
same = logical
   1
```

Limitations

The following functions, which support custom floating-point inputs, do not support complex custom floating-point inputs.

- `ceil`
- `cosh`
- `exp`
- `fix`
- `floor`
- `ge`
- `gt`
- `hypot`
- `le`
- `log`
- `log10`
- `log2`
- `lt`
- `mod`
- `pow10`
- `pow2`
- `power`
- `rem`
- `round`
- `rsqrt`
- `sqrt`
- `tanh`

Version History

Introduced in R2020a

See Also

`half` | `single` | `double`

Topics

"Floating-Point Numbers"

DataTypeWorkflow.findDecoupledSubsystems

Get a list of subsystems to replace with an approximation

Syntax

```
systemsToApproximate = DataTypeWorkflow.findDecoupledSubsystems(system)
```

Description

`systemsToApproximate = DataTypeWorkflow.findDecoupledSubsystems(system)` returns a table containing all of the subsystems in the system specified by `system` created by the Fixed-Point Tool during the preparation stage of conversion.

When converting a model to fixed point using the Fixed-Point Tool, when you click **Prepare**, the tool finds any blocks that are not supported for conversion. When the tool finds these blocks, it isolates the block by placing it in a subsystem surrounded by Data Type Conversion blocks. After converting the rest of the system to fixed point, use this function to get a list of all the subsystems you must replace. You can use the Lookup Table Optimizer to generate a lookup table approximation of the subsystems containing the unsupported blocks.

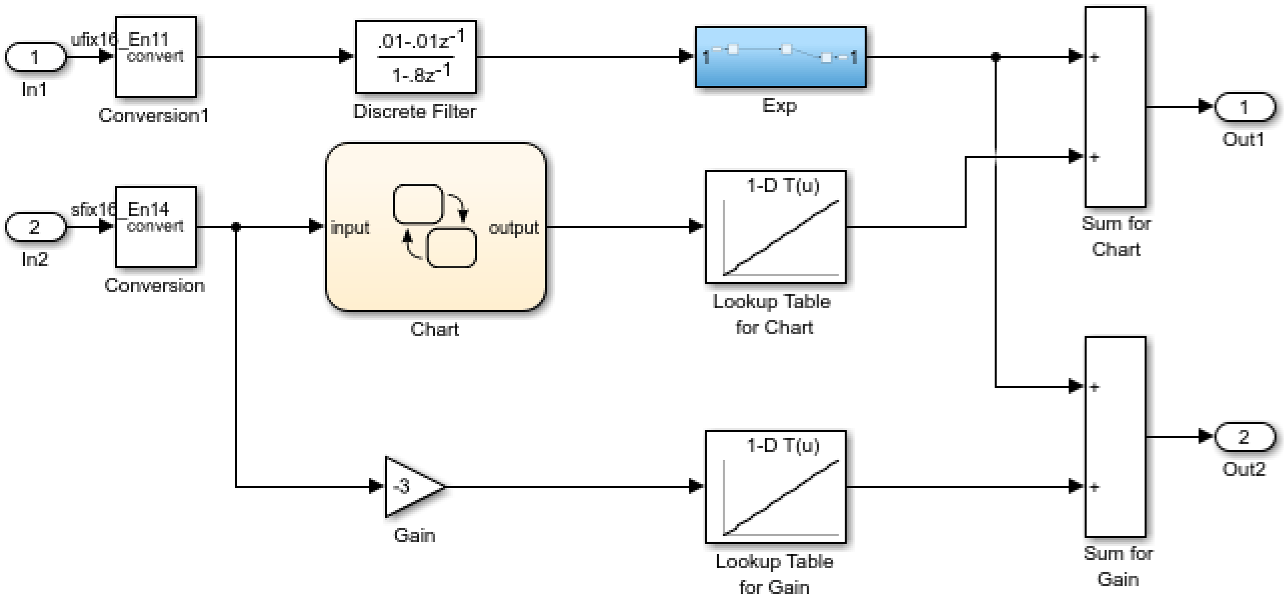
Examples

Replace Unsupported Blocks with a Lookup Table Approximation

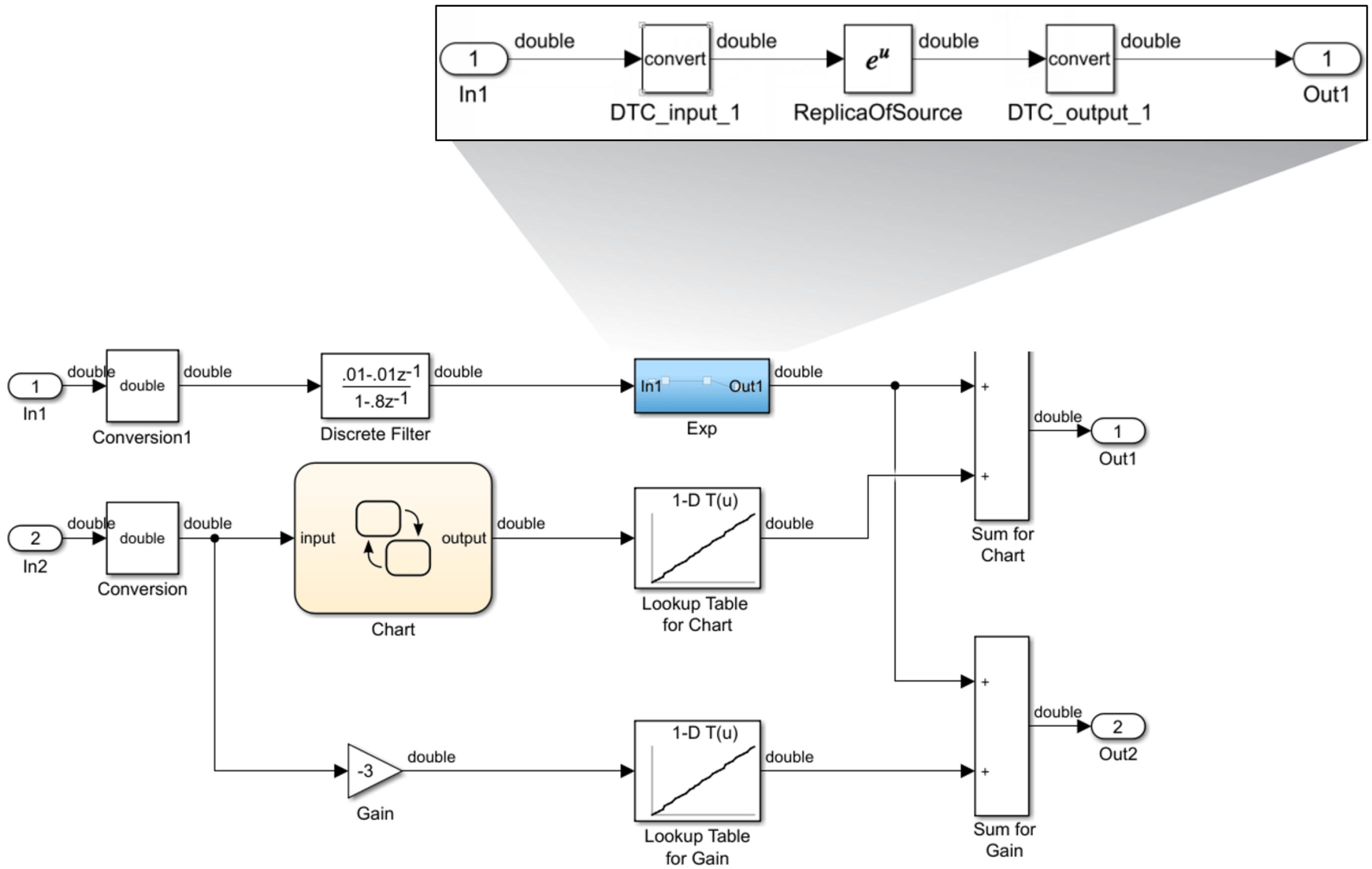
In this example, you replace a block that is not supported for fixed-point conversion, with a lookup table approximation.

Open the model.

```
open_system('ex_fixed_point_workflow_lutapprox')
```



The Controller Subsystem in the model uses fixed-point data types, except in the Exp subsystem. This subsystem was created by the Fixed-Point Tool during the preparation stage of the conversion. In this example, you use the Lookup Table Optimizer to replace this subsystem with a lookup table approximation.



Identify the subsystems that you need to replace using the `DataTypeWorkflow.findDecoupledSubsystems` function.

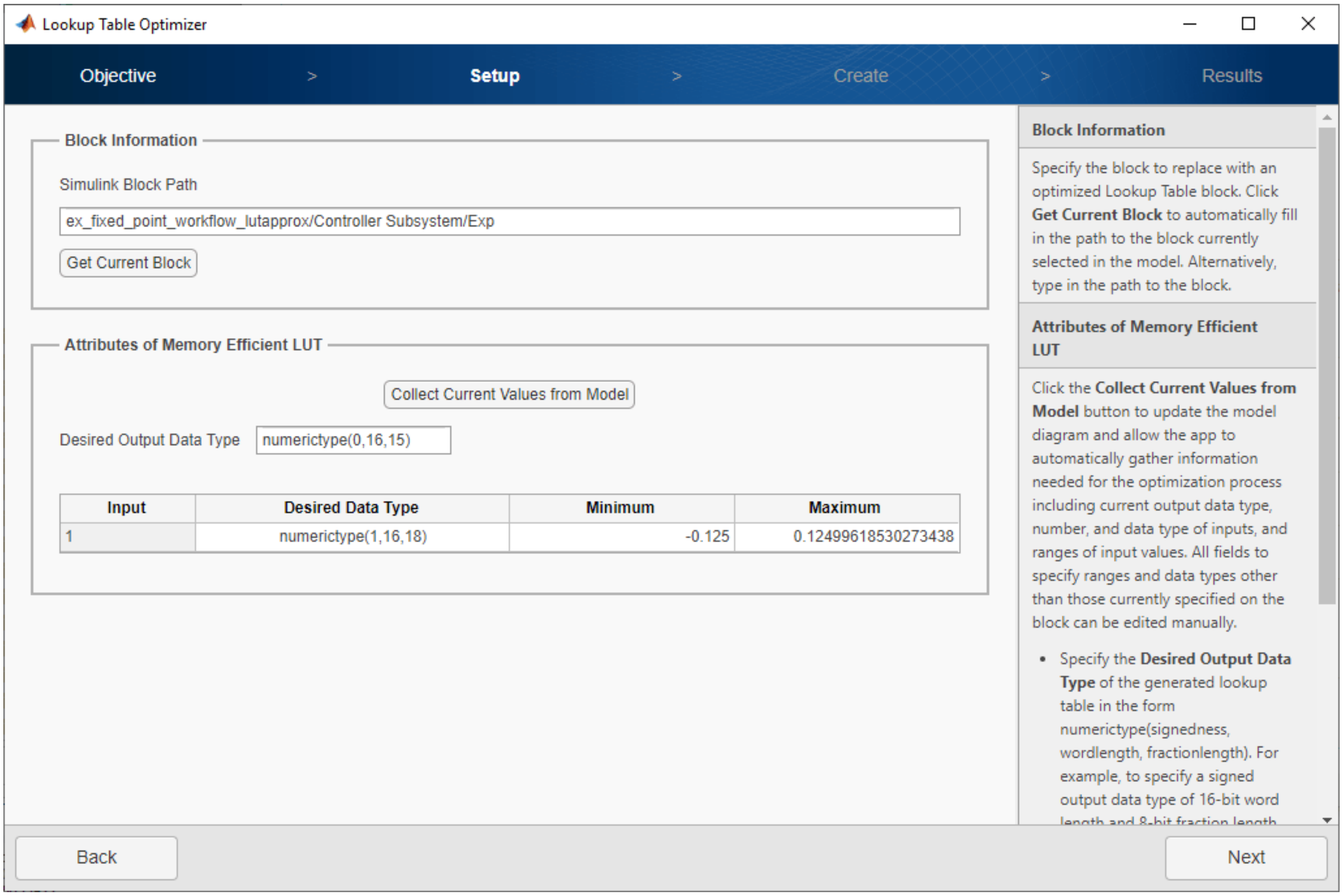
```
decoupled = DataTypeWorkflow.findDecoupledSubsystems(gcs)
```

```
decoupled =

  1x2 table

    ID                             BlockPath
    __    _____________________________________________________________

    1     {'ex_fixed_point_workflow_lutapprox/Controller Subsystem/Exp'}
```

To replace the functions, open the Lookup Table Optimizer. In the Simulink **Apps** tab, select **Lookup Table Optimizer**.

On the **Objective** page of the Lookup Table Optimizer, select **Simulink Block**. Click **Next**.

Under **Block Information**, copy and paste the path to the decoupled subsystem created by the Fixed-Point Tool.



Continue through the steps of the Lookup Table Optimizer to generate the lookup table approximation.

Input Arguments

`system` — System containing the decoupled subsystems

character vector

System containing the decoupled subsystems, specified as a character vector.

Output Arguments

`systemsToApproximate` — Subsystems to approximate with a lookup table

table

A list of the subsystems decoupled from the model by the Fixed-Point Tool to approximate, returned as a table.

Version History

Introduced in R2019a

See Also

`DataTypeWorkflow.Converter` | **Lookup Table Optimizer**

Topics

"Convert Floating-Point Model to Fixed Point"
"Use the Fixed-Point Tool to Prepare a System for Conversion"

dec

Package: `embedded`

Unsigned decimal representation of stored integer of `fi` object

Syntax

`b = dec(a)`

Description

`b = dec(a)` returns the stored integer of `fi` object `a` in unsigned decimal format as a character vector.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

Examples

View Stored Integer of `fi` Object in Unsigned Decimal Format

Create a signed `fi` object with values `-1` and `1`, a word length of 8 bits, and a fraction length of 7 bits.

```
a = fi([-1 1], 1, 8, 7)
```

```
a =
   -1.0000    0.9922

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Find the unsigned decimal representation of the stored integers of `fi` object `a`.

```
b = dec(a)
```

```
b =
'128   127'
```

Input Arguments

a — Input array
`fi` object

Input array, specified as a `fi` object.

Data Types: `fi`

Version History

Introduced before R2006a

See Also

`bin` | `hex` | `storedInteger` | `oct` | `sdec` | `dec2hex` | `dec2base` | `dec2bin`

dec2base

Package: `embedded`

Convert decimal integer to its base-*n* representation for `fi` objects

Syntax

```
baseStr = dec2base(D,n)
baseStr = dec2base(D,n,minDigits)
```

Description

`baseStr = dec2base(D,n)` returns a base-*n* representation of the decimal integer `D`. The output argument `baseStr` is a character array that represents digits using numeric characters, and, when `n` is greater than 10, letters. For example, if `n` is 12, the `dec2base` represents the numbers 9, 10, and 11 using the characters `9`, `A`, and `B`, and represents the number 12 as the character sequence `10`.

`baseStr = dec2base(D,n,minDigits)` returns a base-*n* representation of `D` with no fewer than `minDigits` digits.

Tip `dec2base` returns the base-*n* representation of the real-world value of the values contained in `fi` object `D`.

Examples

Convert Decimal Number

Convert a decimal number to a character vector that represents its value in base 3.

```
D = fi(23);
baseStr = dec2base(D,3)
```

```
baseStr =

    '212'
```

Convert a decimal number to a character vector that represents its value in base 12. In this base system, the characters `'A'` and `'B'` represent the numbers denoted as `10` and `11` in base 10.

```
D = fi(23);
baseStr = dec2base(D,12)
```

```
baseStr =

    '1B'
```

Specify Number of Digits

Specify the number of base-3 digits that `dec2base` returns. If you specify more digits than are required, then `dec2base` pads the output with leading zeros.

```
D = fi(23);
baseStr = dec2base(D,3,5)
```

```
baseStr =

    '00212'
```

Convert Upperbound of `fi` Object

Convert the upper bound of a signed `fi` object with 100-bit word length to base 36 representation.

```
baseStr = dec2base(upperbound(fi([],1,100,0)),36)
```

```
baseStr =

    '1PG7OTO50BLAOIQ8FPQ7'
```

Input Arguments

D — Input array
`fi` array of nonnegative numbers

Input array, specified as a `fi` array of nonnegative numbers.

`D` must contain finite integers. If any element of `D` has a fractional part, then `dec2base` produces an error. For example, `dec2base(fi(10),8)` converts `fi(10)` to `'12'`, but `dec2base(fi(10.5),8)` produces an error.

Data Types: `fi`

n — Base of output representation
integer between 2 and 36

Base of output representation, specified as an integer between 2 and 36. For example, if `n` is `8`, then the output represents base-8 numbers.

`minDigits` — Minimum number of digits in output
positive integer

Minimum number of digits in the output, specified as a positive integer.

- If `D` can be represented with fewer than `minDigits` digits, then `dec2base` pads the output with leading zeros.
- If `D` is so large that it must be represented with more than `minDigits` digits, then `dec2base` returns the output with as many digits as required.

Version History

Introduced in R2021b

Extended Capabilities

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported.

See Also

`fi` | `dec2bin` | `dec2hex` | `bin` | `dec` | `oct` | `hex`

dec2bin

Package: embedded

Convert decimal integer to its binary representation for fi objects

Syntax

```
binStr = dec2bin(D)
binStr = dec2bin(D,minDigits)
```

Description

binStr = dec2bin(D) returns the binary, or base-2, representation of the decimal integer D. The output argument binStr is a character vector that represents binary digits using the characters 0 and 1.

binStr = dec2bin(D,minDigits) returns a binary representation with no fewer than minDigits digits.

Tip dec2bin returns the binary representation of the real-world value of the fi object D. To obtain the binary representation of the stored integer value, use bin instead.

Examples

Convert Decimal Number

Convert a decimal number stored as a fi object to a character vector that represents its binary value.

```
D1 = fi(2748);
D2 = fi(251);
binStr1 = dec2bin(D1)
binStr2 = dec2bin(D2)
```

```
binStr1 =

    '101010111100'


binStr2 =

    '11111011'
```

The dec2bin function converts negative numbers using their two's complement binary values.

```
D3 = fi(-5);
binStr3 = dec2bin(D3)
```

```
binStr3 =

    '11111011'
```

Specify Minimum Number of Digits

Convert the decimal number stored as a `fi` object to binary representation. Specify the minimum number of binary digits that `dec2bin` returns. If you specify more digits than are required, then `dec2bin` pads the output.

```
D = fi(2748);
binStr = dec2bin(D,16)
```

```
binStr =

    '0000101010111100'
```

If you specify fewer digits, then `dec2bin` still returns as many binary digits as required to represent the input number.

```
binStr = dec2bin(D,8)
```

```
binStr =

    '101010111100'
```

Convert Numeric Array

Create a numeric `fi` array.

```
D = fi([1023 122 14]);
```

To represent the elements of `D` as binary values, use the `dec2bin` function. Each row of `binStr` corresponds to an element of `D`.

```
binStr = dec2bin(D)
```

```
binStr =

  3×10 char array

    '1111111111'
    '0001111010'
    '0000001110'
```

Convert Upper and Lower Bound of `fi` Object

Convert the upper and lower bound of a signed `fi` object with 100-bit word length.

```
binStr = dec2bin([lowerbound(fi([],1,100,0)),...
    upperbound(fi([],1,100,0))])
```

```
binStr =
```

```
2×100 char array

  '1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  '0111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

Input Arguments

D — Input array
numeric fi array

Input array, specified as a numeric `fi` array.

- `D` must contain finite integers. If any element of `D` has a fractional part, then `dec2bin` truncates it before conversion. For example, `dec2bin` converts both `fi(12)` and `fi(12.5)` to `'1100'`. The truncation is always to the nearest integer less than or equal to that element.
- `D` can include negative numbers. The function converts negative numbers using their two's complement binary values.

Data Types: `fi`

`minDigits` — Minimum number of digits in output
positive integer

Minimum number of digits in the output, specified as a positive integer.

- If `D` can be represented with fewer than `minDigits` binary digits, then `dec2bin` pads the output.
- If `D` is so large that it must be represented with more than `minDigits` digits, then `dec2bin` returns the output with as many digits as required.

Version History

Introduced in R2021b

Extended Capabilities

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported.

See Also

`fi` | `bin` | `dec2base` | `dec2hex` | `bin` | `dec` | `oct` | `hex`

dec2hex

Package: `embedded`

Convert decimal integer to its hexadecimal representation for `fi` objects

Syntax

```
hexStr = dec2hex(D)
hexStr = dec2hex(D,minDigits)
```

Description

`hexStr = dec2hex(D)` returns the hexadecimal, or base-16, representation of the decimal integer `D`. The output argument `hexStr` is a character array where each row represents the hexadecimal digits of each decimal integer in `D` using the characters `0-9` and `A-F`. `D` must contain finite integers.

`hexStr = dec2hex(D,minDigits)` returns a hexadecimal representation with no fewer than `minDigits` digits.

Tip `dec2hex` returns the hexadecimal representation of the real-world value of the `fi` object `D`. To obtain the hexadecimal representation of the stored integer value, use `hex` instead.

Examples

Convert Decimal Number

Convert the decimal number stored as a `fi` object to hexadecimal representation.

```
D1 = fi(2748);
D2 = fi(251);
hexStr1 = dec2hex(D1)
hexStr2 = dec2hex(D2)
```

```
hexStr1 =

    'ABC'


hexStr2 =

    'FB'
```

The `dec2hex` function converts negative numbers using their two's complement binary values.

```
D3 = fi(-5);
hexStr3 = dec2hex(D3)
```

```
hexStr3 =

    'FB'
```

Specify Minimum Number of Digits

Convert the decimal number stored as a `fi` object to hexadecimal representation. Specify the minimum number of hexadecimal digits that `dec2hex` returns. If you specify more digits than are required, then `dec2hex` pads the output.

```
D = fi(2748);
hexStr = dec2hex(D,8)
```

```
hexStr =

    '00000ABC'
```

If you specify fewer digits, then `dec2hex` still returns as many hexadecimal digits as required to represent the input number.

```
hexStr = dec2hex(D,2)
```

```
hexStr =

    'ABC'
```

Convert Numeric Array

Create a numeric `fi` array.

```
D = fi([1023 122 14]);
```

To represent the elements of `D` as hexadecimal values, use the `dec2hex` function. Each row of `hexStr` corresponds to an element of `D`.

```
hexStr = dec2hex(D)
```

```
hexStr =

  3×3 char array

    '3FF'
    '07A'
    '00E'
```

Convert a numeric `fi` array containing negative values and specify minimum number of digits.

```
D = fi([1023 122 14;2748 251 -5]);
hexStr = dec2hex(D,5)
```

```
hexStr =

  6×5 char array

    '003FF'
```

```
    '00ABC'
    '0007A'
    '000FB'
    '0000E'
    'FFFFB'
```

Convert Upper and Lower Bound of fi Object

Convert the upper and lower bound of a signed `fi` object with 100-bit word length.

```
binStr = dec2hex([lowerbound(fi([],1,100,0)),...
    upperbound(fi([],1,100,0))])
```

```
binStr =

  2×25 char array

    '8000000000000000000000000'
    '7FFFFFFFFFFFFFFFFFFFFFFFF'
```

Input Arguments

D — Input array
numeric `fi` array

Input array, specified as a numeric `fi` array.

- `D` must contain finite integers. If any element of `D` has a fractional part, then `dec2hex` produces an error. For example, `dec2hex` converts `fi(10)` to `'A'`, but does not convert `fi(10.5)`.
- `D` can include negative numbers. The function converts negative numbers using their two's complement binary values.

Data Types: `fi`

minDigits — Minimum number of digits in output
positive integer

Minimum number of digits in the output, specified as a positive integer.

- If `D` can be represented with fewer than `minDigits` hexadecimal digits, then `dec2hex` pads the output.
- If `D` is so large that it must be represented with more than `minDigits` digits, then `dec2hex` returns the output with as many digits as required.

Version History

Introduced in R2021b

Extended Capabilities

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported.

See Also

fi | dec2base | dec2bin | hex | bin | dec | oct

denormalmax

Largest denormalized quantized number for `quantizer` object

Syntax

```
x = denormalmax(q)
```

Description

`x = denormalmax(q)` is the largest positive denormalized quantized number where `q` is a `quantizer` object. Anything larger than `x` is a normalized number. Denormalized numbers apply only to floating-point format. When `q` represents fixed-point numbers, this function returns `eps(q)`.

Examples

```
q = quantizer('float',[6 3]);
x = denormalmax(q)

x =

    0.1875
```

Algorithms

When `q` is a floating-point `quantizer` object,

```
denormalmax(q) = realmin(q) - denormalmin(q)
```

When `q` is a fixed-point `quantizer` object,

```
denormalmax(q) = eps(q)
```

Version History

Introduced before R2006a

See Also

`denormalmin` | `eps` | `quantizer`

denormalmin

Smallest denormalized quantized number for `quantizer` object

Syntax

```
x = denormalmin(q)
```

Description

`x = denormalmin(q)` is the smallest positive denormalized quantized number where `q` is a `quantizer` object. Anything smaller than `x` underflows to zero with respect to the `quantizer` object `q`. Denormalized numbers apply only to floating-point format. When `q` represents a fixed-point number, `denormalmin` returns `eps(q)`.

Examples

```
q = quantizer('float',[6 3]);
x = denormalmin(q)

x =

    0.0625
```

Algorithms

When `q` is a floating-point `quantizer` object,

$$x = 2^{E_{min} - f}$$

where E_{min} is equal to `exponentmin(q)`.

When `q` is a fixed-point `quantizer` object,

$$x = \mathrm{eps}(q) = 2^{-f}$$

where f is equal to `fractionlength(q)`.

Version History

Introduced before R2006a

See Also

`denormalmax` | `eps` | `quantizer`

divide

Package: `embedded`

Divide two `fi` objects

Syntax

```
c = divide(T,a,b)
```

Description

`c = divide(T,a,b)` performs division on the elements of `a` by the elements of `b`. The result `c` has the numeric type specified by `numerictype` object `T`.

Examples

Divide Two fi Objects

This example shows how to control the precision of the `divide` function.

Create an unsigned `fi` object with an 80-bit word length and 2^-83 scaling, which puts the leading 1 of the representation into the most significant bit. Initialize the object with value 0.1, and examine the binary representation.

```
P = fipref('NumberDisplay', 'bin',...
    'NumericTypeDisplay', 'short',...
    'FimathDisplay', 'none');
a = fi(0.1, 0, 80, 83)
```

```
a = 
11001100110011001100110011001100110011001100110011010000000000000000000000000000
      numerictype(0,80,83)
```

Notice that the infinite repeating representation is truncated after 52 bits, because the mantissa of an IEEE® standard double-precision floating-point number has 52 bits.

Contrast the above to calculating 1/10 in fixed-point arithmetic with the quotient set to the same numeric type as before.

```
T = numerictype('Signed', false,...
    'WordLength', 80,...
    'FractionLength', 83);
a = fi(1);
b = fi(10);
c = divide(T, a, b);
c.bin
```

```
ans = 
'11001100110011001100110011001100110011001100110011001100110011001100110011001101'
```

Notice that when you use the `divide` function, the quotient is calculated to the full 80 bits, regardless of the precision of `a` and `b`. Thus, the `fi` object `c` represents 1/10 more precisely than a IEEE® standard double-precision floating-point number can.

Input Arguments

T — Numeric type of the output

`numerictype` object

Numeric type of the output, specified as a `numerictype` object.

a — Numerator

scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array.

Inputs `a` and `b` must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

b — Denominator

scalar | vector | matrix | multidimensional array

Denominator, specified as a real scalar, vector, matrix, or multidimensional array.

Inputs `a` and `b` must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

Output Arguments

c — Quotient

scalar | vector | matrix | multidimensional array

Solution, returned as a scalar, vector, matrix, or multidimensional array.

The size of `c` is determined by implicit expansion of the dimensions of `a` and `b`. For more information, see “Compatible Array Sizes for Basic Operations”.

Algorithms

If `a` and `b` are both `fi` objects, `c` has the same `fimath` object as `a`. If `c` has a `fi` `Fixed` data type, and any one of the inputs have `fi` floating point data types, then the `fi` floating point is converted into a fixed-point value. Intermediate quantities are calculated using the `fimath` object of `a`.

If either a or b is a fi object, and the other is a MATLAB built-in numeric type, then the built-in object is cast to the word length of the fi object, preserving best-precision fraction length. Intermediate quantities are calculated using the fimath object of the input fi object.

If a and b are both MATLAB built-in doubles, then c is the floating-point quotient a./b, and numerictype T is ignored.

Data Type Propagation Rules

For syntaxes for which Fixed-Point Designer software uses the numerictype object T, the divide function follows the data type propagation rules listed in the following table. In most cases, floating-point data types are propagated. This allows you to write code that can be used with both fixed-point and floating-point inputs.

Data Type of Input fi Objects a and b		Data Type of numerictype Object T	Data Type of Output c
Built-in double	Built-in double	Any	Built-in double
fi Fixed	fi Fixed	fi Fixed	Data type of numerictype object T
fi Fixed	fi Fixed	fi double	fi double
fi Fixed	fi Fixed	fi single	fi single
fi Fixed	fi Fixed	fi ScaledDouble	fi ScaledDouble with properties of numerictype object T
fi double	fi double	fi Fixed	fi double
fi double	fi double	fi double	fi double
fi double	fi double	fi single	fi single
fi double	fi double	fi ScaledDouble	fi double
fi single	fi single	fi Fixed	fi single
fi single	fi single	fi double	fi double
fi single	fi single	fi single	fi single
fi single	fi single	fi ScaledDouble	fi single
fi ScaledDouble	fi ScaledDouble	fi Fixed	If either input a or b is of type fi ScaledDouble, then output cis of type fi ScaledDouble with properties of numerictype object T.
fi ScaledDouble	fi ScaledDouble	fi double	fi double
fi ScaledDouble	fi ScaledDouble	fi single	fi single

Data Type of Input fi Objects a and b		Data Type of numerictype Object T	Data Type of Output c
fi ScaledDouble	fi ScaledDouble	fi ScaledDouble	If either input a or b is of type fi ScaledDouble, then output c is of type fi ScaledDouble with properties of numerictype object T.

Version History

Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators

Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for `fi divide`, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
- Complex and imaginary divisors are not supported.
- Code generation does not support the syntax `T.divide(a,b)`.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

- For HDL Code generation, the divisor must be a constant and a power of two.
- Non-`fi` inputs must be constant; that is, their values must be known at compile time so that they can be cast to `fi` objects.
- Complex and imaginary divisors are not supported.
- Code generation in MATLAB does not support the syntax `T.divide(a,b)`.

See Also

add | fi | fimath | mpy | mrdivide | numerictype | rdivide | sub | sum

double

Double-precision floating-point real-world value of `fi` object

Syntax

`b = double(a)`

Description

`b = double(a)` returns the real-world value of a `fi` object in double-precision floating point format.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

Examples

View Stored Integer of `fi` Object in Double-Precision Format

Create a signed `fi` object with values `-1` and `1`, a word length of 8 bits, and a fraction length of 7 bits.

```
a = fi([-1 1], 1, 8, 7)
```

```
a =
   -1.0000    0.9922

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Find the double-precision floating-point real-world value of the stored integers of `fi` object `a`.

```
b = double(a)
```

```
b = 1×2

   -1.0000    0.9922
```

Input Arguments

a — `fi` object to view in double-precision floating-point
`fi` object

Input `fi` object to view in double-precision floating-point.

Data Types: `fi`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts. Instead of using casts, supply a replacement function. For more information, see “Function Replacements”.

See Also

`single`

eps

Quantized relative accuracy for `fi` or `quantizer` objects

Syntax

```
d = eps(a)
d = eps(q)
```

Description

`d = eps(a)` returns the value of the least significant bit value of the `fi` object `a`. The result of this function is equivalent to that given by the Fixed-Point Designer function `lsb`.

`d = eps(q)` returns the value of the least significant bit of the value of the `quantizer` object `q`.

Examples

Quantized Relative Accuracy of `fi` Object

```
a = fi(pi, 1, 8)
eps(a)
```

```
ans =

    0.1250
```

Quantization Level of `quantizer` Object

```
q = quantizer('fixed',[6 3]);
eps(q)
```

```
ans =

    0.1250
```

Input Arguments

a — Input `fi` object
`fi` object

Input `fi` object.

Data Types: `fi`

q — Input `quantizer` object
`quantizer` object

Input `quantizer` object.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation supports scalar fixed-point signals only.
- Code generation supports scalar, vector, and matrix, `fi` single and `fi` double signals.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

- Supported for scalar fixed-point signals only.
- Supported for scalar, vector, and matrix, `fi` single and `fi` double signals.

See Also
`intmax` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `upperbound` | `quantizer` | `fi`

eq, ==

Package: `embedded`

Determine whether real-world values are equal

Syntax

```
A == B
eq(A,B)
```

Description

`A == B` returns a logical array with elements set to logical `1` (`true`) where the real-world values of arrays `A` and `B` are equal, when `A` or `B` is a `fi` object. Otherwise, the element is logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to a fixed-point type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object.

`eq(A,B)` is an alternate way to execute `A == B`, but is rarely used.

Examples

Compare Two `fi` Objects

Use the eq function to determine if two `fi` objects have the same real-world value.

```
a = fi(pi);
b = fi(pi,1,32);
a == b
```

```
ans = logical
   0
```

Input `a` has a 16-bit word length, while input `b` has a 32-bit word length. The `eq` function returns `0` because the two `fi` objects do not have the same real-world value.

Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the floating-point double is cast to a type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

```
a = fi(pi);
b = pi;
eq(a,b)

ans =

  logical

   0
```

Input Arguments

A,B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Version History

Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators
Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for fi eq, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a try/catch block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

R2022a: Improved accuracy in comparing fi objects and floating-point numbers using relational operators
Behavior changed in R2022a

In previous releases, when comparing a single or double to a fi object, the floating-point value was cast to the same word length and signedness of the fi object. This could lead to incorrect results. For example,

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   0   0
```

```
fi(65534)
fi(65534.25) == 65534.25

ans = 

      65534

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: -1

ans =

  logical

   1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   1   0
```

Note that the updated algorithm may produce subtle, but accurate, results. For example:

```
fi(pi) == pi

ans =

  logical

   0
```

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest wordlength growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`ge` | `gt` | `isequal` | `le` | `lt` | `ne`

errmean

Package: `embedded`

Mean of quantization error

Syntax

```
m = errmean(q)
```

Description

`m = errmean(q)` returns the mean of a uniformly distributed random quantization error that arises from quantizing a signal by `quantizer` object `q`.

Note The results are not exact when the signal precision is close to the precision of the `quantizer`.

Examples

Mean of Quantization Error

Find the mean of the quantization error for the `quantizer` object `q`.

```
q = quantizer;
m = errmean(q)
```

```
m = -1.5259e-05
```

Compare this result to the sample mean from a Monte Carlo experiment.

```
r = realmax(q);
u = 2*r*rand(1000,1)-r;  % Original signal
y = quantize(q,u);       % Quantized signal
e = y - u;               % Error
m_est = mean(e)          % Estimate of the error mean
```

```
m_est = -1.5028e-05
```

Input Arguments

q — Input `quantizer` object
`quantizer` object

Input `quantizer` object, specified as a `quantizer` object.

Version History

Introduced in R2008a

See Also

errpdf | errvar | quantize

errpdf

Probability density function of quantization error

Syntax

```
[f,x] = errpdf(q)
f = errpdf(q,x)
```

Description

`[f,x] = errpdf(q)` returns the probability density function `f` evaluated at the values in `x`. The vector `x` contains the uniformly distributed random quantization errors that arise from quantizing a signal by `quantizer` object `q`.

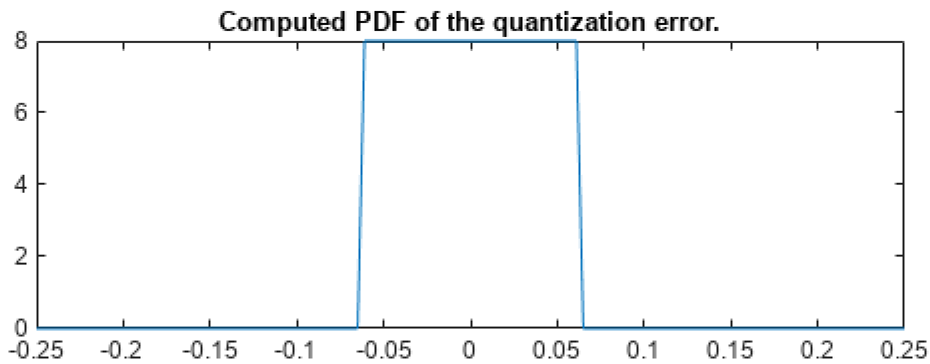
`f = errpdf(q,x)` returns the probability density function `f` evaluated at the values in vector `x`.

Note The results are not exact when the signal precision is close to the precision of the `quantizer`.

Examples

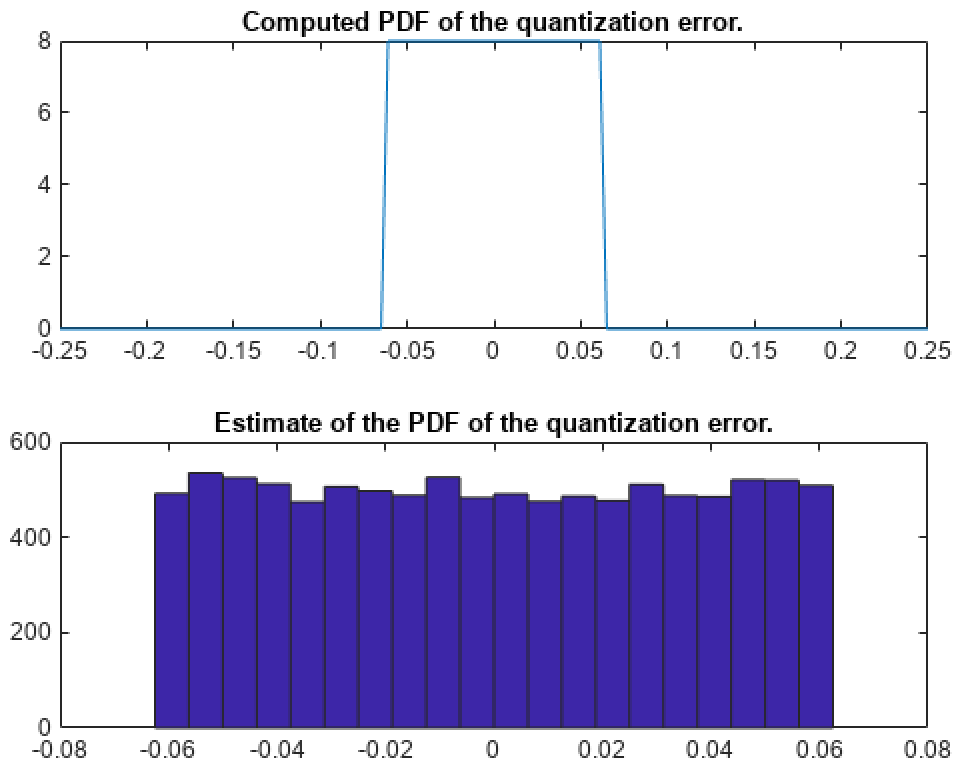
Compute the PDF of the quantization error

```
q = quantizer('nearest',[4 3]);
[f,x] = errpdf(q);
subplot(211)
plot(x,f)
title('Computed PDF of the quantization error.')
```



The output plot shows the probability density function of the quantization error. Compare this result to a plot of the sample probability density function from a Monte Carlo experiment:

```
r = realmax(q);
    u = 2*r*rand(10000,1)-r;  % Original signal
    y = quantize(q,u);        % Quantized signal
    e = y - u;                % Error
    subplot(212)
    hist(e,20)
    gca.xlim = [min(x) max(x)];
    title('Estimate of the PDF of the quantization error.')
```



Version History

Introduced in R2008a

See Also

`errmean` | `errvar` | `quantize`

errvar

Variance of quantization error

Syntax

```
v = errvar(q)
```

Description

`v = errvar(q)` returns the variance of a uniformly distributed random quantization error that arises from quantizing a signal by `quantizer` object `q`.

Note The results are not exact when the signal precision is close to the precision of the `quantizer`.

Examples

Find `v`, the variance of the quantization error for `quantizer` object `q`:

```
q = quantizer;
v = errvar(q)
```

```
v =

     7.761021455128987e-11
```

Now compare `v` to `v_est`, the sample variance from a Monte Carlo experiment:

```
r = realmax(q);
      u = 2*r*rand(1000,1)-r;  % Original signal
      y = quantize(q,u);       % Quantized signal
      e = y - u;               % Error
      v_est = var(e)           % Estimate of the error variance
```

```
v_est =

     7.686538499583834e-11
```

Version History

Introduced in R2008a

See Also

`errmean` | `errpdf` | `quantize`

exponentbias

Exponent bias for `quantizer` object

Syntax

```
b = exponentbias(q)
```

Description

`b = exponentbias(q)` returns the exponent bias of the `quantizer` object `q`. For fixed-point `quantizer` objects, `exponentbias(q)` returns 0.

Examples

```
q = quantizer('double');
b = exponentbias(q)
```

```
b =

        1023
```

Algorithms

For floating-point `quantizer` objects,

$$b = 2^{e-1} - 1$$

where `e` = `eps(q)`, and `exponentbias` is the same as the exponent maximum.

For fixed-point `quantizer` objects, `b = 0` by definition.

Version History

Introduced before R2006a

See Also

`eps` | `exponentlength` | `exponentmax` | `exponentmin`

exponentlength

Exponent length of `quantizer` object

Syntax

`e = exponentlength(q)`

Description

`e = exponentlength(q)` returns the exponent length of `quantizer` object `q`. When `q` is a fixed-point `quantizer` object, `exponentlength(q)` returns `0`. This is useful because exponent length is valid whether the `quantizer` object mode is floating point or fixed point.

Examples

```
q = quantizer('double');
e = exponentlength(q)
```

```
e =

    11
```

Algorithms

The exponent length is part of the format of a floating-point `quantizer` object `[w e]`. For fixed-point `quantizer` objects, *e* = 0 by definition.

Version History

Introduced before R2006a

See Also

`eps` | `exponentbias` | `exponentmax` | `exponentmin`

exponentmax

Maximum exponent for `quantizer` object

Syntax

`exponentmax(q)`

Description

`exponentmax(q)` returns the maximum exponent for `quantizer` object `q`. When `q` is a fixed-point `quantizer` object, it returns `0`.

Examples

```
q = quantizer('double');
exponentmax(q)
```

```
ans =

        1023
```

Algorithms

For floating-point `quantizer` objects,

$$E_{max} = 2^{e-1} - 1$$

For fixed-point `quantizer` objects, $E_{max} = 0$ by definition.

Version History

Introduced before R2006a

See Also

`eps` | `exponentbias` | `exponentlength` | `exponentmin`

exponentmin

Minimum exponent for `quantizer` object

Syntax

`emin = exponentmin(q)`

Description

`emin = exponentmin(q)` returns the minimum exponent for `quantizer` object `q`. If `q` is a fixed-point `quantizer` object, `exponentmin` returns 0.

Examples

```
q = quantizer('double');
emin = exponentmin(q)
```

```
emin =

      -1022
```

Algorithms

For floating-point `quantizer` objects,

$$E_{min} = -2^{e-1} + 2$$

For fixed-point `quantizer` objects, $E_{min} = 0$.

Version History

Introduced before R2006a

See Also

`eps` | `exponentbias` | `exponentlength` | `exponentmax`

eye

Create identity matrix with fixed-point properties

Syntax

```
I = eye('like',p)
I = eye(n,'like',p)
I = eye(n,m,'like',p)
I = eye(sz,'like',p)
```

Description

`I = eye('like',p)` returns the scalar 1 with the same fixed-point properties and complexity (real or complex) as the prototype argument, `p`. The output, `I`, contains the same `numerictype` and `fimath` properties as `p`.

`I = eye(n,'like',p)` returns an `n`-by-`n` identity matrix like `p`, with ones on the main diagonal and zeros elsewhere.

`I = eye(n,m,'like',p)` returns an `n`-by-`m` identity matrix like `p`.

`I = eye(sz,'like',p)` returns an array like `p`, where the size vector, `sz`, defines `size(I)`.

Examples

Create Identity Matrix with Fixed-Point Properties

Create a prototype `fi` object, `p`.

```
p = fi([],1,16,14);
```

Create a 3-by-4 identity matrix with the same fixed-point properties as `p`.

```
I = eye(3,4,'like',p)
```

```
I =
     1     0     0     0
     0     1     0     0
     0     0     1     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14
```

Create Identity Matrix with Attached fimath

Create a signed `fi` object with word length of 16, fraction length of 15 and `OverflowAction` set to `Wrap`.

```
format long
p = fi([],1,16,15,'OverflowAction','Wrap');
```

Create a 2-by-2 identity matrix with the same `numerictype` properties as `p`.

```
X = eye(2,'like',p)
```

```
X =
   0.999969482421875                   0
                   0   0.999969482421875

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

1 cannot be represented by the data type of `p`, so the value saturates. The output `fi` object `X` has the same `numerictype` and `fimath` properties as `p`.

Input Arguments

`n` — Size of first dimension of `I`
integer value

Size of first dimension of `I`, specified as an integer value.

- If `n` is the only integer input argument, then `I` is a square `n`-by-`n` identity matrix.
- If `n` is `0`, then `I` is an empty matrix.
- If `n` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`m` — Size of second dimension of `I`
integer value

Size of second dimension of `I`, specified as an integer value.

- If `m` is `0`, then `I` is an empty matrix.
- If `m` is negative, then it is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`sz` — Size of `I`
row vector of no more than two integer values

Size of `I`, specified as a row vector of no more than two integer values.

- If an element of `sz` is `0`, then `I` is an empty matrix.
- If an element of `sz` is negative, then the element is treated as `0`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

p — Prototype
`fi` object | numeric variable

Prototype, specified as a `fi` object or numeric variable.

If the value 1 overflows the numeric type of `p`, the output saturates regardless of the specified `OverflowAction` property of the attached `fimath`. All subsequent operations performed on the output obey the rules of the attached `fimath`.

Data Types: `fi` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Version History

Introduced in R2015a

See Also

`zeros` | `ones`

Topics

"Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types Using cast and zeros"
"Manual Fixed-Point Conversion Best Practices"

fi

Construct fixed-point numeric object

Description

To assign a fixed-point data type to a number or variable, create a `fi` object using the `fi` constructor. You can specify numeric attributes and math rules in the constructor or by using the `numerictype` and `fimath` objects.

Creation

Syntax

```
a = fi
a = fi(v)
a = fi(v,s)
a = fi(v,s,w)
a = fi(v,s,w,f)
a = fi(v,s,w,slope,bias)
a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias)
a = fi(v,T)
a = fi( ___ ,F)
a = fi( ___ ,Name,Value)
```

Description

`a = fi` returns a signed `fi` object with no value, a 16-bit word length, and a 15-bit fraction length.

`a = fi(v)` returns a signed `fi` object with value `v`, a 16-bit word length, and best-precision fraction length.

`a = fi(v,s)` returns a `fi` object with value `v`, signedness `s`, a 16-bit word length, and best-precision fraction length.

`a = fi(v,s,w)` returns a `fi` object with value `v`, signedness `s`, and word length `w`.

`a = fi(v,s,w,f)` returns a `fi` object with value `v`, signedness `s`, word length `w`, and fraction length `f`.

`a = fi(v,s,w,slope,bias)` returns a `fi` object with value `v`, signedness `s`, `slope`, and `bias`.

`a = fi(v,s,w,slopeadjustmentfactor,fixedexponent,bias)` returns a `fi` object with value `v`, signedness `s`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

`a = fi(v,T)` returns a `fi` object with value `v` and `numerictype` `T`.

`a = fi( ___ ,F)` returns a `fi` object with `fimath` `F`.

`a = fi( ___ ,Name,Value)` returns a `fi` object with property values specified by one or more name-value pair arguments.

Input Arguments

v — Value
scalar | vector | matrix | multidimensional array

Value of the `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

The value of the returned `fi` object is the value of the input `v` quantized to the data type specified in the `fi` constructor. When the input `v` is a non-double and you do not specify the word length or fraction length, the returned `fi` object retains the numerictype of the input. For an example, see "Create fi Object From Non-double Value" on page 4-394.

You can specify the non-finite values `-Inf`, `Inf`, and `NaN` as the value only if you fully specify the numerictype of the `fi` object. When `fi` is specified as a fixed-point numerictype,

- `NaN` maps to `0`.
- When the `'OverflowAction'` property of the `fi` object is set to `'Wrap'`, `-Inf`, and `Inf` map to `0`.
- When the `'OverflowAction'` property of the `fi` object is set to `'Saturate'`, `Inf` maps to the largest representable value, and `-Inf` maps to the smallest representable value.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

s — Signedness
`true` or `1` (default) | `false` or `0`

Signedness of the `fi` object, specified as a numeric or logical `1` (`true`) or `0` (`false`). A value of `1` (`true`) indicates a signed data type. A value of `0` (`false`) indicates an unsigned data type.

Data Types: `logical`

w — Word length in bits
16 (default) | positive scalar integer

Word length in bits of the `fi` object, specified as a positive scalar integer.

The `fi` object has a word length limit of 65535 bits.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

f — Fraction length in bits
15 (default) | scalar integer

Fraction length in bits of the stored integer value of the `fi` object, specified as a scalar integer.

If you do not specify a fraction length, the `fi` object automatically uses the fraction length that gives the best precision while avoiding overflow for the specified value, word length, and signedness.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

slope — Slope
positive scalar

Slope of the scaling of the `fi` object, specified as a positive scalar.

This equation represents the real-world value of a slope bias scaled number.

$$real - worldvalue = (slope \times integer) + bias$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

bias — Bias
scalar

Bias of the scaling of the `fi` object, specified as a scalar.

This equation represents the real-world value of a slope bias scaled number.

$$real - worldvalue = (slope \times integer) + bias$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

slopeadjustmentfactor — Slope adjustment factor
scalar greater than or equal to 1 and less than 2

Slope adjustment factor of the `fi` object, specified as a scalar greater than or equal to `1` and less than `2`.

The following equation demonstrates the relationship between the slope, fixed exponent, and slope adjustment factor.

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

fixedexponent — Fixed exponent
scalar

Fixed exponent of the `fi` object, specified as a scalar.

The following equation demonstrates the relationship between the slope, fixed exponent, and slope adjustment factor.

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

T — Numeric type properties
`numerictype` object

Numeric type properties of the `fi` object, specified as a `numerictype` object.

F — Fixed-point math properties
`fimath` object

Fixed-point math properties of the `fi` object, specified as a `fimath` object.

Properties

The `fi` object has three types of properties:

- fi Object Data Properties
- fimath Object Properties
- numerictype Object Properties

You can set these properties when you create a `fi` object. Use the data properties to access data in a `fi` object. The `fimath` properties and `numerictype` properties are, by transitivity, also properties of the `fi` object. `fimath` properties determine the rules for performing fixed-point arithmetic operations on `fi` objects. The `numerictype` object contains all the data type and scaling attributes of a fixed-point object.

Examples

Create `fi` Object

Create a `fi` object using the default constructor. The constructor returns a signed `fi` object with no value, a 16-bit word length, and a 15-bit fraction length.

```
a = fi
```

```
a =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Create a signed `fi` object with a value of `pi`, a 16-bit word length, and best-precision fraction length. The fraction length is automatically set to achieve the best precision possible without overflow.

```
a = fi(pi)
```

```
a =
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Create an unsigned `fi` object with a value of `pi`. When you specify only the value and the signedness of the `fi` object, the word length defaults to 16 bits with best-precision fraction length.

```
a = fi(pi,0)
```

```
a =
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 14
```

Create a signed `fi` object with a word length of 8 bits and best-precision fraction length. In this example, the fraction length of `a` is `5` because three bits are required to represent the integer portion of the value when the data type is signed.

```
a = fi(pi,1,8)
```

```
a =
    3.1562

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 5
```

If the `fi` object is unsigned, only two bits are needed to represent the integer portion, leaving six fractional bits.

```
b = fi(pi,0,8)
```

```
b =
    3.1406

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 6
```

Create a signed `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits.

```
a = fi(pi,1,8,3)
```

```
a =
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

Create an array of `fi` objects with 16-bit word length and 12-bit fraction length.

```
a = fi((magic(3)/10),1,16,12)
```

```
a =
    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
          WordLength: 16
      FractionLength: 12
```

Create fi Object with Slope and Bias Scaling

The real-world value of a slope and bias scaled number is represented by

$$\text{real world value} = (\text{slope} \times \text{integer}) + \text{bias}.$$

To create a fi object that uses slope and bias scaling, include the slope and bias arguments after the word length in the constructor. For example, create a fi object with a slope of 3 and a bias of 2.

```
a = fi(pi,1,16,3,2)
```

```
a = 
     2

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

The DataTypeMode property of the fi object a is Fixed-point: slope and bias scaling.

Alternatively, you can specify the slope adjustment factor and fixed exponent where

$$slope = \text{slopeadjustmentfactor} \times 2^{\text{fixedexponent}}.$$

For example, create a fi object with a slope adjustment factor of 1.5, a fixed exponent of 1, and a bias of 2.

```
a = fi(pi,1,16,1.5,1,2)
```

```
a = 
     2

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

Create fi Object From numerictype Object

A numerictype object contains all of the data type information of a fi object. numerictype properties are also properties of fi objects.

You can create a fi object that uses all of the properties of an existing numerictype object by specifying the numerictype object in the fi constructor.

```
T = numerictype(0,24,16)
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 24
        FractionLength: 16
```

```
a = fi(pi,T)
```

```
a =
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 24
        FractionLength: 16
```

Create `fi` Object With Associated `fimath`

The arithmetic attributes of a `fi` object are defined by a `fimath` object which is attached to that `fi` object.

Create a `fimath` object and specify the `OverflowAction`, `RoundingMethod`, and `ProductMode` properties.

```
F = fimath('OverflowAction','Wrap',...
    'RoundingMethod','Floor',...
    'ProductMode','KeepMSB')
```

```
F =
        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: KeepMSB
     ProductWordLength: 32
               SumMode: FullPrecision
```

Create a `fi` object and specify the `fimath` object `F` in the constructor.

```
a = fi(pi,F)
```

```
a =
    3.1415

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: KeepMSB
     ProductWordLength: 32
               SumMode: FullPrecision
```

Use the `removefimath` function to remove the associated `fimath` object and restore the math settings to their default values.

```
a = removefimath(a)
```

```
a = 
    3.1415

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Create `fi` Object From Non-double Value

When the input argument `v` of a `fi` object is not a double and you do not specify the word length or fraction length properties, the returned `fi` object retains the numeric type of the input.

Create `fi` object from built-in integer

When the input is a built-in integer, the fixed-point attributes match the attributes of the integer type.

```
v1 = uint32(5);
a1 = fi(v1)
```

```
a1 = 
     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 32
        FractionLength: 0
```

```
v2 = int8(5);
a2 = fi(v2)
```

```
a2 = 
     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0
```

Create `fi` object from `fi` object

When the input value is a `fi` object, the output uses the same word length, fraction length, and signedness as the input `fi` object.

```
v = fi(pi,1,24,12);
a = fi(v)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
```

```
          Signedness: Signed
          WordLength: 24
      FractionLength: 12
```

Create `fi` object from logical

When the input value is a logical, the `DataTypeMode` property of the output `fi` object is `Boolean`.

```
v = true;
a = fi(v)
```

```
a = 
   1

          DataTypeMode: Boolean
```

Create `fi` object from single

When the input value is single, the `DataTypeMode` property of the output is `Single`.

```
v = single(pi);
a = fi(v)
```

```
a = 
    3.1416

          DataTypeMode: Single
```

Specify Rounding and Overflow Modes in `fi` Object Constructor

You can set `fimath` properties, such as rounding and overflow modes during the creation of the `fi` object.

```
a = fi(pi,'RoundingMethod','Floor',...
    'OverflowAction','Wrap')
```

```
a = 
    3.1415

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

The `RoundingMethod` and `OverflowAction` properties are properties of the `fimath` object. Specifying these properties in the `fi` constructor associates a local `fimath` object with the `fi` object.

Use the `removefimath` function to remove the local `fimath` and set the math properties back to their default values.

```
a = removefimath(a)
```

```
a = 
    3.1415

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Set Data Type Override on fi Object

This examples shows how to use the `DataTypeOverride` setting of the `fipref` object to override `fi` objects with doubles, singles, or scaled doubles. The `fipref` object defines the display and logging attributes for all `fi` objects.

Save the current `fipref` settings to restore later.

```
fp = fipref;
initialDTO = fp.DataTypeOverride;
```

Create a `fi` object with the default settings and original `fipref` settings.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Use the `fipref` object to turn on data type override to doubles.

```
fipref('DataTypeOVerride','TrueDoubles')
```

```
ans = 
                NumberDisplay: 'RealWorldValue'
           NumericTypeDisplay: 'full'
                FimathDisplay: 'full'
                  LoggingMode: 'Off'
             DataTypeOverride: 'TrueDoubles'
    DataTypeOverrideAppliesTo: 'AllNumericTypes'
```

Create a new `fi` object without specifying its `DataTypeOverride` property so that it uses the data type override settings specified using `fipref`.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Double
```

Create another `fi` object and set its `DataTypeOverride` setting to `off` so that it ignores the data type override settings of the `fipref` object.

```
b = fi(pi,'DataTypeOverride','Off')
```

```
b = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Restore the `fipref` settings saved at the start of the example.

```
fp.DataTypeOverride = initialDTO;
```

fi Behavior for -Inf, Inf, and NaN

To use the non-numeric values `-Inf`, `Inf`, and `NaN` as fixed-point values with `fi`, you must fully specify the numeric type of the fixed-point object. Automatic best-precision scaling is not supported for these values.

Saturate on Overflow

When the numeric type of the `fi` object is specified to saturate on overflow, then `Inf` maps to the largest representable value of the specified numeric type, and `-Inf` maps to the smallest representable value. `NaN` maps to zero.

```
x = [-inf nan inf];
a = fi(x,1,8,0,'OverflowAction','Saturate')
b = fi(x,0,8,0,'OverflowAction','Saturate')
```

```
a = 

  -128     0   127

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision

b = 

     0     0   255

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0

        RoundingMethod: Nearest
```

```
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Wrap on Overflow

When the numeric type of the `fi` object is specified to wrap on overflow, then `-Inf`, `Inf`, and `NaN` map to zero.

```
x = [-inf nan inf];
a = fi(x,1,8,0,'OverflowAction','Wrap')
b = fi(x,0,8,0,'OverflowAction','Wrap')
```

```
a =

     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision

b =

     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Tips

- Use the `fipref` object to control the display, logging, and data type override preferences for `fi` objects.

Version History

Introduced before R2006a

R2020b: Change in default behavior of `fi` for `-Inf`, `Inf`, and `NaN`

Behavior changed in R2020b

In previous releases, `fi` would return an error when passed the non-finite input values `-Inf`, `Inf`, or `NaN`. `fi` now treats these inputs in the same way that MATLAB and Simulink handle `-Inf`, `Inf`, and `NaN` for integer data types.

When `fi` is specified as a fixed-point numeric type,

- `NaN` maps to `0`.
- When the `'OverflowAction'` property of the `fi` object is set to `'Wrap'`, `-Inf`, and `Inf` map to `0`.
- When the `'OverflowAction'` property of the `fi` object is set to `'Saturate'`, `Inf` maps to the largest representable value, and `-Inf` maps to the smallest representable value.

For an example of this behavior, see “fi Behavior for -Inf, Inf, and NaN” on page 4-397.

Note Best-precision scaling is not supported for input values of `-Inf`, `Inf`, or `NaN`.

R2021a: Inexact property names for `fi`, `fimath`, and `numerictype` objects not supported

In previous releases, inexact property names for `fi`, `fimath`, and `numerictype` objects would result in a warning. In R2021a, support for inexact property names was removed. Use exact property names instead.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The default constructor syntax without any input arguments is not supported.
- If the `numerictype` is not fully specified, the input to `fi` must be a constant, a `fi`, a single, or a built-in integer value. If the input is a built-in double value, it must be a constant. This limitation allows `fi` to autoscale its fraction length based on the known data type of the input.
- All properties related to data type must be constant for code generation.
- `numerictype` object information must be available for nonfixed-point Simulink inputs.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`fimath` | `numerictype` | `fipref`

Topics
“Create Fixed-Point Data”
“Perform Fixed-Point Arithmetic”
“Perform Binary-Point Scaling”
“Binary Point Interpretation”

fiaccel

Accelerate fixed-point code or convert floating-point MATLAB code to fixed-point MATLAB code

Syntax

```
fiaccel -options fcn
fiaccel -float2fixed fcn
```

Description

`fiaccel -options fcn` translates the MATLAB file `fcn.m` to a MEX function, which accelerates fixed-point code. To use `fiaccel`, your code must meet one of these requirements:

- The top-level function has no inputs or outputs, and the code uses `fi`.
- The top-level function has an output or a non-constant input, and at least one output or input is a `fi`.
- The top-level function has at least one input or output containing a built-in integer class (`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, or `uint64`), and the code uses `fi`.

`fiaccel -float2fixed fcn` converts the floating-point MATLAB function, `fcn` to fixed-point MATLAB code.

Examples

Accelerate Fixed-Point MATLAB Code

This example shows how to accelerate fixed-point MATLAB code using the `fiaccel` function.

Define a function that computes the moving average.

```
type("moving_average.m")
```

```
function [avg,z] = moving_average(x,z)
%#codegen

if nargin < 2
    z = fi(zeros(10,1),1,16,15);
end


z(2:end) = z(1:end-1);  % Update buffer
z(1) = x;               % Add new value
avg = mean(z);          % Compute moving average
end
```

Create a test file.

```
type("test_moving_average.m")
```

```
function avg = test_moving_average(x)
%#codegen
```

```
if nargin < 1
    x = fi(rand(100,1),1,16,15);
end

z = fi(zeros(10,1),1,16,15);
avg = x;
for k = 1:length(x)
    [avg(k),z] = moving_average(x(k),z);
end
end
```

Use the `fiaccel` function to create a MEX function and accelerate the MATLAB code.

```
x = fi(rand(100,1),1,16,15);
fiaccel test_moving_average -args {x} -report
```

Compare the non-accelerated and accelerated code.

```
tic avg = test_moving_average(x); toc       % Non-compiled version
tic avg = test_moving_average_mex(x); toc   % Compiled version
```

Convert Floating-Point MATLAB Code to Fixed Point

Create a `coder.FixptConfig` object, `fixptcfg`, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is `dti_test`.

```
fixptcfg.TestBenchName = 'dti_test';
```

Convert a floating-point MATLAB function to fixed-point MATLAB code. In this example, the MATLAB function name is `dti`.

```
fiaccel -float2fixed fixptcfg dti
```

Input Arguments

`fcn` — MATLAB function to generate MEX from
function name

MATLAB function to generate MEX from, specified as a function existing in the current working folder or on the path.

Note If your top-level file is on a path that contains Unicode characters, code generation might not be able to find the file.

`-options` — Compiler options
option value | space delimited list of option values

Compiler options, specified as a space delimited list of option values. `fiaccel` gives precedence to individual command-line options over options specified using a configuration object. If command-line options conflict, the right-most option prevails.

Specified as one or more of these values:

`-args` *`example_inputs`*	Define the size, class, and complexity of MATLAB function inputs by providing a cell array of example input values. The position of the example input in the cell array must correspond to the position of the input argument in the MATLAB function definition. To generate a function that has fewer input arguments than the function definition has, omit the example values for the arguments that you do not want. Specify the example inputs immediately after the function to which they apply. Instead of an example value, you can provide a `coder.Type` object. To create a `coder.Type` object, use the `coder.typeof` function.
`-config` *`config_object`*	Specify MEX generation parameters, based on *`config_object`*, defined as a MATLAB variable using `coder.mexconfig`. For example: `cfg = coder.mexconfig;`
`-d` *`out_folder`*	Store generated files in the absolute or relative path specified by *`out_folder`*. If the folder specified by *`out_folder`* does not exist, `fiaccel` creates it for you. If you do not specify the folder location, `fiaccel` generates files in the default folder `fiaccel/`*`mex/fcn`*, where *`fcn`* is the name of the MATLAB function specified at the command line. The function does not support the following characters in folder names: asterisk (*), question-mark (?), dollar ($), and pound (#).

`-float2fixed` *`float2fixed_cfg_name`*	Generates fixed-point MATLAB code using the settings specified by the floating-point to fixed-point conversion configuration object named *`float2fixed_cfg_name`*. For this option, `fiaccel` generates files in the folder `codegen/fcn_name/fixpt`. You must set the `TestBenchName` property of *`float2fixed_cfg_name`*. For example: `fixptcfg.TestBenchName = 'myadd_test';` specifies that `myadd_test` is the test file for the floating-point to fixed-point configuration object `fixptcfg`. You cannot use this option with the `-global` option.
`-g`	Compiles the MEX function in debug mode, with optimization turned off. If not specified, `fiaccel` generates the MEX function in optimized mode.
`-global` *`global_values`*	Specify initial values for global variables in MATLAB file. Use the values in cell array `global_values` to initialize global variables in the function you compile. The cell array should provide the name and initial value of each global variable. You must initialize global variables before compiling with `fiaccel`. If you do not provide initial values for global variables using the `-global` option, `fiaccel` checks for the variable in the MATLAB global workspace. If you do not supply an initial value, `fiaccel` generates an error. The generated MEX code and MATLAB each have their own copies of global data. To ensure consistency, you must synchronize their global data whenever the two interact. If you do not synchronize the data, their global variables might differ. You cannot use this option with the `-float2fixed` option.
`-I` *`include_path`*	Add *`include_path`* to the beginning of the code generation path. `fiaccel` searches the code generation path *first* when converting MATLAB code to MEX code.

`-launchreport`	Generate and open a code generation report. If you do not specify this option, `fiaccel` generates a report only if error or warning messages occur or you specify the `-report` option.
`-nargout`	Specify the number of output arguments in the generated entry-point function. The code generator produces the specified number of output arguments in the order in which they occur in the MATLAB function definition.
`-o` *`output_file_name`*	Generate the MEX function with the base name *`output_file_name`* plus a platform-specific extension. *`output_file_name`* can be a file name or include an existing path. If you do not specify an output file name, the base name is *`fcn_mex`*, which allows you to run the original MATLAB function and the MEX function and compare the results.
`-O` *`optimization_option`*	Optimize generated MEX code, based on the value of *`optimization_option`*: • `enable:inline` — Enable function inlining • `disable:inline` — Disable function inlining If not specified, `fiaccel` uses inlining for optimization.
`-report`	Generate a code generation report. If you do not specify this option, `fiaccel` generates a report only if error or warning messages occur or you specify the `-launchreport` option.
`-?`	Display help for `fiaccel` command.

Version History

Introduced in R2011a

R2023a: Change to default `fiaccel` behavior for constant inputs

When the `fiaccel` function generates a MEX file, it no longer automatically removes constants from the call to the MEX file. With this change, `fiaccel` and `codegen` MEX generation now have the same default behavior with respect to constant inputs.

```
fiaccel myfun -args {x,coder.Constant(c)}
myfun_mex(x,c)
```

To revert to the prior behavior, set the `ConstantInputs` property of the `coder.MexConfig` object to `'Remove'`.

```
cfg = coder.MexConfig;
cfg.ConstantInputs = "Remove";
fiaccel myfun -args {x,coder.Constant(c)} -config cfg
myfun_mex(x)
```

See Also

coder.ArrayType | coder.Constant | coder.EnumType | coder.FiType | coder.newtype | coder.PrimitiveType | coder.resize | coder.StructType | coder.Type | coder.typeof | coder.mexconfig | coder.config | coder.FixPtConfig

filter

Package: `embedded`

1-D digital filter of `fi` objects

Syntax

```
y = filter(b,1,x)
[y,zf] = filter(b,1,x,zi)
y = filter(b,1,x,zi,dim)
```

Description

`y = filter(b,1,x)` filters the data in the fixed-point vector `x` using the filter described by the fixed-point vector `b`. The function returns the filtered data in the output `fi` object `y`.

`filter` always operates along the first non-singleton dimension. Thus, the filter operates along the first dimension for column vectors and nontrivial matrices and along the second dimension for row vectors.

`[y,zf] = filter(b,1,x,zi)` uses initial conditions `zi` for the filter delays. The length of `zi` must equal `length(b)-1`. The final conditions of the delays are returned in `zf`.

`y = filter(b,1,x,zi,dim)` acts along dimension `dim`. If you do not want to specify the vector of initial conditions, use `[]` for the input argument `zi`.

Note This function is a 1-D digital filter for `fi` objects. To filter non-`fi` data, use the MATLAB `filter` function.

Examples

Filter High-Frequency Fixed-Point Sinusoid From Signal

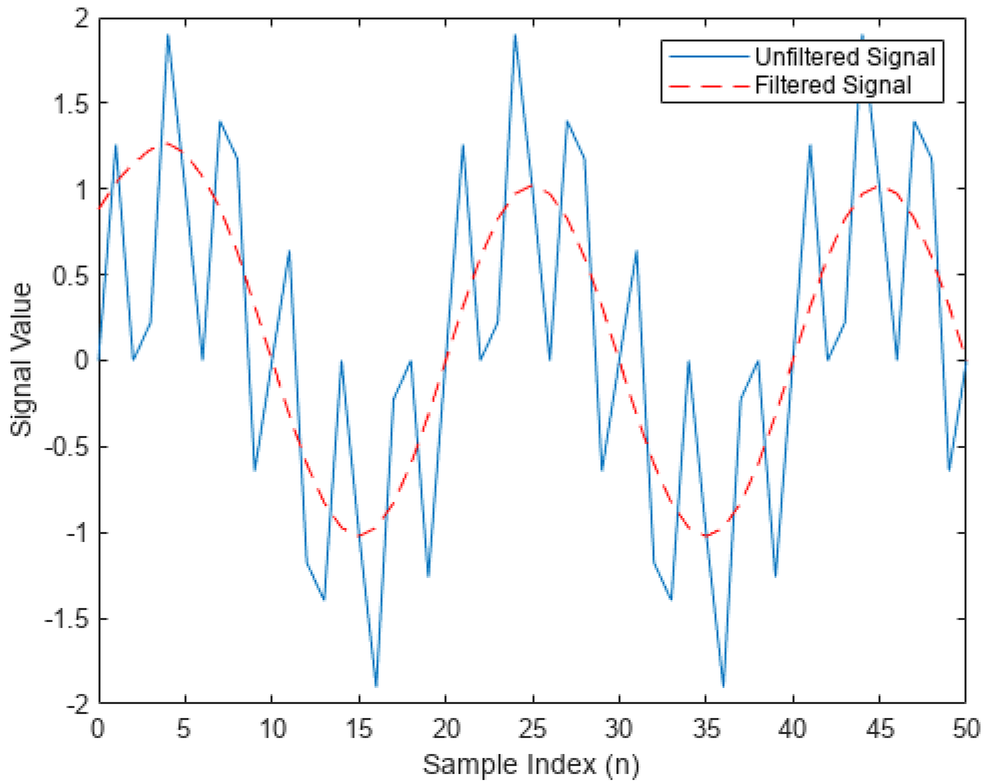
Filter a high-frequency fixed-point sinusoid from a signal that contains both a low- and high-frequency fixed-point sinusoid.

```
w1 = 0.1*pi;
w2 = 0.6*pi;
n  = 0:999;
xd = sin(w1*n) + sin(w2*n);
x  = sfi(xd,12);
b  = ufi([0.1:0.1:1,1-0.1:-0.1:0.1]/4,10);
gd = (length(b)-1)/2;
y  = filter(b,1,x);
```

Plot the results, accomodating for the group delay of the filter.

```
plot(n(1:end-gd),x(1:end-gd))
hold on
```

```
plot(n(1:end-gd),y(gd+1:end),'r--')
axis([0 50 -2 2])
legend('Unfiltered Signal','Filtered Signal')
xlabel('Sample Index (n)')
ylabel('Signal Value')
```



The resulting plot shows both the unfiltered and filtered signals.

Input Arguments

b — Filter coefficients

fixed-point vector

Filter coefficients, specified as a fixed-point vector.

Data Types: `fi`

x — Input data

fixed-point vector

Input data, specified as a fixed-point vector.

Data Types: `fi`

zi — Initial conditions for filter delays

[] (default) | fixed-point vector | fixed-point matrix | fixed-point multidimensional array

Initial conditions for filter delays, specified as a fixed-point vector. `zi` must be a `fi` object with the same data type as `y` and `zf`.

- If `zi` is a vector, then its length must be `length(b)-1`.
- If `zi` is a matrix or multidimensional array, then the size of the leading dimension must be `length(b)-1`. The size of each remaining dimension must match the size of the corresponding dimension of `x`.

If you do not specify a value for `zi`, or if you specify `[]`, it defaults to a fixed-point array with a value of `0` and the appropriate `numerictype` and size.

Data Types: `fi`

`dim` — Dimension along which to operate
positive integer scalar

Dimension along which to operate, specified as a positive integer scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`y` — Filtered data
fixed-point vector | fixed-point matrix | fixed-point multidimensional array

Filtered data, returned as a fixed-point `fi` vector, matrix, or multidimensional array.

`zf` — Final conditions for filter delays
vector | matrix | multidimensional array

Final conditions for filter delays, returned as a fixed-point `fi` vector, matrix, or multidimensional array.

Tips

- The `filter` function only supports FIR filters. In the general filter representation *b*/*a*, the denominator *a* of an FIR filter is the scalar 1, which is the second input of this function.
- The `numerictype` of `b` can be different than the `numerictype` of `x`.
- If you want to specify initial conditions but do not know what `numerictype` to use, first try filtering your data without initial conditions. You can do so by specifying `[]` for the input `zi`. After performing the filtering operation, you have the `numerictype` of `y` and `zf` (if requested). Because the `numerictype` of `zi` must match that of `y` and `zf`, you now know the `numerictype` to use for the initial conditions.

Algorithms

Filter length (*L*)

The filter length is `length(b)` or the number of filter coefficients specified in the fixed-point vector `b`.

Filter order (*N*)

The filter order is the number of states (delays) of the filter and is equal to `L-1`.

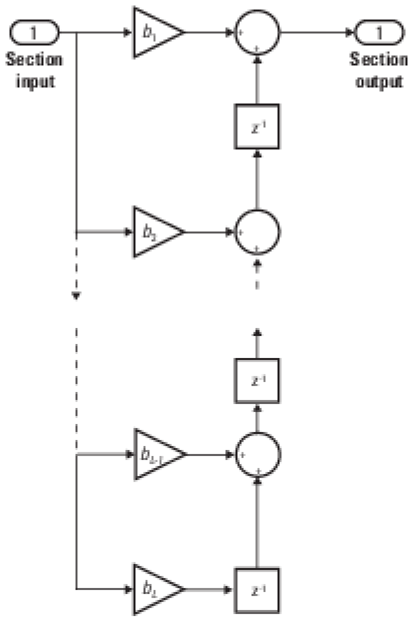
Direct-Form Transposed FIR Filter

The `filter` function uses a Direct-Form Transposed FIR implementation of this difference equation:

$$y(n) = b_1 * x_n + b_2 * x_{n-1} + \ldots + b_L * x_{n-N}$$

where L is the "Filter length (L)" on page 4-408 and N is the "Filter order (N)" on page 4-409.

This diagram shows the direct-form transposed FIR filter structure used by the `filter` function.



`fimath` Propagation Rules

The `filter` function uses these rules regarding `fimath` behavior:

- `globalfimath` is obeyed.
- If any of the inputs has an attached `fimath`, then it is used for intermediate calculations.
- If more than one input has an attached `fimath`, then the `fimaths` must be equal.
- The output `y` is always associated with the default `fimath`.
- If the input vector `zi` has an attached `fimath`, then the output vector `zf` retains this `fimath`.

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `Specify precision` or `Keep LSB`.

See Also

`conv` | `filter`

fimath

Set fixed-point math settings

Syntax

```
F = fimath
F = fimath(Name,Value)
```

Description

`F = fimath` creates a `fimath` object with the default `fimath` property settings.

`F = fimath(Name,Value)` specifies the properties of a `fimath` object by using one or more name-value pair arguments. All properties not specified in the constructor use default values.

Examples

Create a Default fimath Object

This example shows how to create a `fimath` object with the default property settings.

```
F = fimath
```

```
F =
        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Set Properties of a fimath Object

Set the properties of a `fimath` object at the time of object creation by using name-value pairs. For example, set the overflow action to saturate and the rounding method to convergent.

```
F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent')
```

```
F =
        RoundingMethod: Convergent
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = fimath('OverflowAction','Saturate','RoundingMethod','Floor')`

`CastBeforeSum` — Whether both operands are cast to the sum data type before addition
`false` or `0` (default) | `true` or `1`

Whether both operands are cast to the sum data type before addition, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Note This property is hidden when the `SumMode` is set to `FullPrecision`.

Example: `F = fimath('CastBeforeSum',true)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`MaxProductWordLength` — Maximum allowable word length for the product data type
`65535` (default) | positive integer

Maximum allowable word length for the product data type, specified as a positive integer.

Example: `F = fimath('MaxProductWordLength',16)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`MaxSumWordLength` — Maximum allowable word length for sum data type
65535 (default) | positive integer

Maximum allowable word length for the sum data type, specified as a positive integer.

Example: `F = fimath('MaxSumWordLength',16)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`OverflowAction` — Action to take on overflow
`'Saturate'` (default) | `'Wrap'`

Action to take on overflow, specified as one of these values:

- `'Saturate'` – Saturate to the maximum or minimum value of the fixed-point range on overflow.
- `'Wrap'` – Wrap on overflow. This mode is also known as two's complement overflow.

Example: `F = fimath('OverflowAction','Wrap')`

Data Types: `char`

`ProductBias` — Bias of product data type
`0` (default) | floating-point number

Bias of the product data type, specified as a floating-point number.

Example: `F = fimath('ProductBias',1)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProductFixedExponent` — Fixed exponent of product data type
`-30` (default) | nonzero integer

Fixed exponent of the product data type, specified as a nonzero integer.

Note The `ProductFractionLength` is the negative of the `ProductFixedExponent`. Changing one property changes the other.

Example: `F = fimath('ProductFixedExponent',-20)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProductFractionLength` — Fraction length of product data type
`30` (default) | nonzero integer

Fraction length, in bits, of the product data type, specified as a nonzero integer.

Note The `ProductFractionLength` is the negative of the `ProductFixedExponent`. Changing one property changes the other.

Example: `F = fimath('ProductFractionLength',20)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProductMode` — How product data type is determined
`'FullPrecision'` (default) | `'KeepLSB'` | `'KeepMSB'` | `'SpecifyPrecision'`

How the product data type is determined, specified as one of these values:

- `'FullPrecision'` – The full precision of the result is kept.
- `'KeepLSB'` – Keep the least significant bits. Specify the product word length. The fraction length is set to maintain the least significant bits of the product.
- `'KeepMSB'` – Keep the most significant bits. Specify the product word length. The fraction length is set to maintain the most significant bits of the product.
- `'SpecifyPrecision'` – Specify the word and fraction lengths or slope and bias of the product.

Example: `F = fimath('ProductMode','KeepLSB')`

Data Types: `char`

`ProductSlope` — Slope of product data type
`9.3132e-10` (default) | finite, positive floating-point number

Slope of the product data type, specified as a finite, positive floating-point number.

Note

$$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$$

Changing one of these properties affects the others.

Example: `F = fimath('ProductSlope',9.3132e-10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProductSlopeAdjustmentFactor` — Slope adjustment factor of the product data type
`1` (default) | floating-point number greater than or equal to 1 and less than 2

Slope adjustment factor of the product data type, specified as a floating-point number greater than or equal to 1 and less than 2.

Note

$$ProductSlope = ProductSlopeAdjustmentFactor \times 2^{ProductFixedExponent}$$

Changing one of these properties affects the others.

Example: `F = fimath('ProductSlopeAdjustmentFactor',1)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ProductWordLength` — Word length of product data type
`32` (default) | positive integer

Word length, in bits, of the product data type, specified as a positive integer.

Example: `F = fimath('ProductWordLength',64)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`RoundingMethod` — Rounding method to use
`'Nearest'` (default) | `'Ceiling'` | `'Convergent'` | `'Zero'` | `'Floor'` | `'Round'`

Rounding method to use, specified as one of these values:

- `'Nearest'` – Round toward nearest. Ties round toward positive infinity.
- `'Ceiling'` – Round toward positive infinity.
- `'Convergent'` – Round toward nearest. Ties round to the nearest even stored integer (least biased).
- `'Zero'` – Round toward zero.
- `'Floor'` – Round toward negative infinity.
- `'Round'` – Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

Example: `F = fimath('RoundingMethod','Convergent')`

Data Types: `char`

`SumBias` — Bias of sum data type
`0` (default) | floating-point number

Bias of the sum data type, specified as a floating-point number.

Example: `F = fimath('SumBias',0)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SumFixedExponent` — Fixed exponent of sum data type
`-30` (default) | nonzero integer

Fixed exponent of the sum data type, specified as a nonzero integer.

Note The `SumFractionLength` is the negative of the `SumFixedExponent`. Changing one property changes the other.

Example: `F = fimath('SumFixedExponent',-20)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SumFractionLength` — Fraction length of sum data type
`30` (default) | nonzero integer

Fraction length, in bits, of the sum data type, specified as a nonzero integer.

Note The `SumFractionLength` is the negative of the `SumFixedExponent`. Changing one property changes the other.

Example: `F = fimath('SumFractionLength',20)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SumMode` — How the sum data type is determined
`'FullPrecision'` (default) | `'KeepLSB'` | `'KeepMSB'` | `'SpecifyPrecision'`

How the sum data type is determined, specified as one of these values:

- `'FullPrecision'` – The full precision of the result is kept.
- `'KeepLSB'` – Keep least significant bits. Specify the sum data type word length. The fraction length is set to maintain the least significant bits of the sum.
- `'KeepMSB'` – Keep most significant bits. Specify the sum data type word length. The fraction length is set to maintain the most significant bits of the sum and no more fractional bits than necessary.
- `'SpecifyPrecision'` – Specify the word and fraction lengths or slope and bias of the sum data type.

Example: `F = fimath('SumMode','KeepLSB')`

Data Types: `char`

`SumSlope` — Slope of sum data type
`9.3132e-10` (default) | floating-point number

Slope of the sum data type, specified as a floating-point number.

Note

$$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$$

Changing one of these properties affects the others.

Example: `F = fimath('SumSlope',9.3132e-10)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SumSlopeAdjustmentFactor` — Slope adjustment factor of the sum data type
1 (default) | floating-point number greater than or equal to 1 and less than 2

Slope adjustment factor of the sum data type, specified as a floating-point number greater than or equal to 1 and less than 2.

Note

$$SumSlope = SumSlopeAdjustmentFactor \times 2^{SumFixedExponent}$$

Changing one of these properties affects the others.

Example: `F = fimath('SumSlopeAdjustmentFactor',1)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SumWordLength` — Word length of sum data type
32 (default) | positive integer

Word length, in bits, of the sum data type, specified as a positive integer.

Example: `F = fimath('SumWordLength',64)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced before R2006a

R2021a: Inexact property names for `fi`, `fimath`, and `numerictype` objects not supported

In previous releases, inexact property names for `fi`, `fimath`, and `numerictype` objects would result in a warning. In R2021a, support for inexact property names was removed. Use exact property names instead.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a `fimath` object. You define this object in the MATLAB Function block dialog in the Model Explorer.
- Use to create `fimath` objects in the generated code.
- If the `ProductMode` property of the `fimath` object is set to anything other than `FullPrecision`, the `ProductWordLength` and `ProductFractionLength` properties must be constant.
- If the `SumMode` property of the `fimath` object is set to anything other than `FullPrecision`, the `SumWordLength` and `SumFractionLength` properties must be constant.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`fi` | `fipref` | `globalfimath` | `numerictype` | `quantizer` | `removefimath` | `setfimath`

Topics
"fimath Object Construction"
"fimath Object Properties"
How Functions Use fimath
"fimath Properties Usage for Fixed-Point Arithmetic"

fipref

Set fixed-point preferences

Syntax

```
P = fipref
P = fipref(Name,Value)
```

Description

`P = fipref` creates a default `fipref` object. The `fipref` object defines the display and logging attributes for all `fi` objects.

`P = fipref(Name,Value)` creates a `fipref` object with properties specified by `Name,Value` pairs.

Your `fipref` settings persist throughout your MATLAB session. Use `reset(fipref)` to return to the default settings during your session. Use `savefipref` to save your display preferences for subsequent MATLAB sessions.

Examples

Create a Default `fipref` Object

```
P = fipref
```

```
P =
         NumberDisplay: 'RealWorldValue'
    NumericTypeDisplay: 'full'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
      DataTypeOverride: 'ForceOff'
```

Set `fipref` Properties at Object Creation

You can set properties of `fipref` objects at the time of object creation by including properties after the arguments of the `fipref` constructor function. For example, to set `NumberDisplay` to `bin` and `NumericTypeDisplay` to `short`,

```
P = fipref('NumberDisplay','bin','NumericTypeDisplay','short')
```

```
P =
         NumberDisplay: 'bin'
    NumericTypeDisplay: 'short'
         FimathDisplay: 'full'
           LoggingMode: 'Off'
```

```
DataTypeOverride: 'ForceOff'
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `P = fipref('NumberDisplay','RealWorldValue','NumericTypeDisplay','short');`

Data Type Override Properties

`DataTypeOverride` — Data type override options
`'ForceOff'` (default) | `'ScaledDoubles'` | `'TrueDoubles'` | `'TrueSingles'`

Data type override options for `fi` objects, specified as the comma-separated pair consisting of `'DataTypeOverride'` and one of these values:

- `'ForceOff'` — No data type override
- `'ScaledDoubles'` — Override with scaled doubles
- `'TrueDoubles'` — Override with doubles
- `'TrueSingles'` — Override with singles

Data type override only occurs when the `fi` constructor function is called.

Data Types: `char`

`DataTypeOverrideAppliesTo` — Data type override setting applicability
`'AllNumericTypes'` (default) | `'Fixed-Point'` | `'Floating-Point'`

Data type override setting applicability to `fi` objects, specified as the comma-separated pair consisting of `'DataTypeOverrideAppliesTo'` and one of these values:

- `'AllNumericTypes'` — Apply data type override to all `fi` data types. This setting does not override built-in integer types.
- `'Fixed-Point'` — Apply data type override only to fixed-point data types
- `'Floating-Point'` — Apply data type override only to floating-point `fi` data types

`DataTypeOverrideAppliesTo` displays only if `DataTypeOverride` is not set to `ForceOff`.

Data Types: `char`

Display Properties

`FimathDisplay` — Display options for local `fimath` attributes of `fi` objects
`'full'` (default) | `'none'`

Display options for the local `fimath` attributes of a `fi` object, specified as the comma-separated pair consisting of `'FimathDisplay'` and one of these values:

- `'full'` — Displays all of the `fimath` attributes of a fixed-point object
- `'none'` — None of the `fimath` attributes are displayed

Data Types: `char`

NumberDisplay — Display options for the value of a fi object
`'RealWorldValue'` (default) | `'bin'` | `'dec'` | `'hex'` | `'int'` | `'none'`

Display options for the values of a `fi` object, specified as the comma-separated pair consisting of `'NumberDisplay'` and one of these values:

- `'bin'` — Displays the stored integer value in binary format
- `'dec'` — Displays the stored integer value in unsigned decimal format
- `'RealWorldValue'` — Displays the stored integer value in the format specified by the MATLAB `format` function

  `fi` objects in `rat` format are displayed according to

  $$\frac{1}{\left(2^{fixed\,-\,pointexponent}\right)} \times storedinteger$$
- `'hex'` — Displays the stored integer value in hexadecimal format
- `'int'` — Displays the stored integer value in signed decimal format
- `'none'` — No value is displayed

The stored integer value does not change when you change the `fipref` object. The `fipref` object only affects the display.

Data Types: `char`

NumericTypeDisplay — Display options for the numerictype attributes of a fi object
`'full'` (default) | `'none'` | `'short'`

Display options for the `numerictype` attributes of a `fi` object, specified as the comma-separated pair consisting of `'NumericTypeDisplay'` and one of these values:

- `'full'` — Displays all of the `numerictype` attributes of a `fi` object
- `'none'` — None of the `numerictype` attributes are displayed
- `'short'` — Displays the `numerictype` attributes of a `fi` object using the abbreviated notation of the `numerictype` constructor

Data Types: `char`

Logging Properties

LoggingMode — Logging options for operations performed on fi objects
`'off'` (default) | `'on'`

Logging options for operations performed on `fi` objects, specified as the comma-separated pair consisting of `'LoggingMode'` and one of these values:

- `'off'` — No logging
- `'on'` — Information is logged for future operations

Overflows and underflows for assignment, plus, minus, and multiplication operations are logged as warnings when `LoggingMode` is set to `on`.

When `LoggingMode` is `on`, you can also use the following functions to return logged information about assignment and creation operations to the MATLAB command line:

- `maxlog` — Returns the maximum real-world value
- `minlog` — Returns the minimum value
- `noverflows` — Returns the number of overflows
- `nunderflows` — Returns the number of underflows

`LoggingMode` must be set to `on` before you perform any operation in order to log information about it. To clear the log, use the function `resetlog`.

Data Types: `char`

Version History

Introduced before R2006a

See Also

`fi` | `fimath` | `numerictype` | `quantizer` | `savefipref`

fix

Round toward zero

Syntax

`y = fix(a)`

Description

`y = fix(a)` rounds `fi` object `a` to the nearest integer in the direction of zero and returns the result in `fi` object `y`.

Examples

Use `fix` on a Signed `fi` Object

The following example demonstrates how the `fix` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi,1,8,3)
```

```
a = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

```
y = fix(a)
```

```
y = 
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 5
        FractionLength: 0
```

The following example demonstrates how the `fix` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a = 
    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
y = fix(a)
```

```
y = 
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Compare Rounding Methods

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

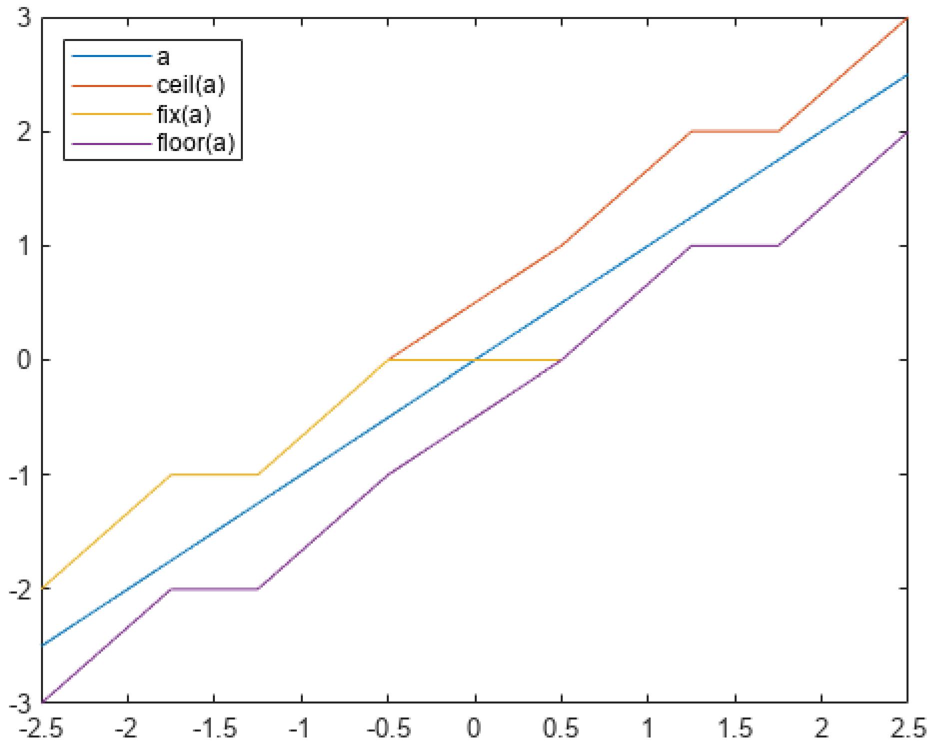
- The `ceil` function rounds values to the nearest integer toward positive infinity.
- The `fix` function rounds values to the nearest integer toward zero.
- The `floor` function rounds values to the nearest integer toward negative infinity.

This example illustrates these differences for a given `fi` input object `a`.

```
a = fi([-2.5,-1.75,-1.25,-0.5,0.5,1.25,1.75,2.5]');
y = [a ceil(a) fix(a) floor(a)]
```

```
y = 
   -2.5000   -2.0000   -2.0000   -3.0000
   -1.7500   -1.0000   -1.0000   -2.0000
   -1.2500   -1.0000   -1.0000   -2.0000
   -0.5000         0         0   -1.0000
    0.5000    1.0000         0         0
    1.2500    2.0000    1.0000    1.0000
    1.7500    2.0000    1.0000    1.0000
    2.5000    3.0000    2.0000    2.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
plot(a,y); legend('a','ceil(a)','fix(a)','floor(a)','location','NW');
```



Input Arguments

a — Input `fi` array

scalar | vector | matrix | multidimensional array

Input `fi` array, specified as scalar, vector, matrix, or multidimensional array.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`fix` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Data Types: `fi`
Complex Number Support: Yes

Algorithms

- `y` and `a` have the same `fimath` object and `DataType` property.
- When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

- When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

Version History

Introduced in R2008a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`ceil` | `convergent` | `floor` | `nearest` | `round`

fixed.extractNumericType

Extract numeric type from input

Syntax

```
T = fixed.extractNumericType(x)
```

Description

`T = fixed.extractNumericType(x)` returns an `embedded.numerictype` object that is extracted from a numeric value input `x`, or is specified by the input argument `x`.

Examples

Extract Numeric Type

Extract the numeric type from an input numeric value.

```
T = fixed.extractNumericType(pi)
```

```
T =


          DataTypeMode: Double
```

```
T = fixed.extractNumericType(int8(0))
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0
```

```
T = fixed.extractNumericType(fi(pi,1,24,12))
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

```
T = fixed.extractNumericType(half(pi))
```

```
T =


          DataTypeMode: Half
```

Extract the numeric type from a numeric type specification object.

```
T = fixed.extractNumericType(numerictype(1,32,16))
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16
```

```
T = fixed.extractNumericType(fixdt(0,18,0))
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 18
        FractionLength: 0
```

Extract the numeric type from a data type name string.

```
T = fixed.extractNumericType('int8')
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0
```

```
T = fixed.extractNumericType('sfix16_En3')
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 3
```

Extract the numeric type from a constructor string.

```
T = fixed.extractNumericType('numerictype(1,33,55)')
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 33
        FractionLength: 55
```

```
T = fixed.extractNumericType('fixdt(0,77,22)')
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
```

```
          Signedness: Unsigned
          WordLength: 77
      FractionLength: 22
```

Input Arguments

x — Input
scalar

Input, specified as a scalar.

The following input types are supported:

- Numeric values — `half`, `single`, `double`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `logical`, `fi`
- Numeric type specification objects — `embedded.numerictype` objects, `Simulink.NumericType` objects
- MATLAB data type name strings — `'half'`, `'single'`, `'double'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, `'uint64'`, `'logical'`
- Simulink data type name strings (not aliases) — `'bool'`, `'sfix16_En3'`, etc.
- Constructor strings that evaluate to a numeric type object — `'numerictype(1,33,55)'`, `'fixdt(0,77,22)'`, etc.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

Output Arguments

T — Numeric type of input
`embedded.numerictype` object

Numeric type of the input, returned as a `embedded.numerictype` object.

Version History

Introduced in R2021a

See Also

`fi` | `fixdt` | `numerictype` | `Simulink.NumericType` | "Fixed-Point Numbers in Simulink"

fixDiv

Round the result of division toward zero

Syntax

```
y = fixDiv(x,d)
y = fixDiv(x,d,m)
```

Description

`y = fixDiv(x,d)` returns the result of `x/d` rounded to the nearest integer value in the direction of zero.

`y = fixDiv(x,d,m)` returns the result of `x/d` rounded to the nearest multiple of `m` in the direction of zero.

The datatype of `y` is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of `x`, and the values of `d` and `m`.

Examples

Divide and Round to Zero

Perform a division operation and round to the nearest integer value in the direction of zero.

```
fixDiv(int16(201),10)
```

```
ans =
    20

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Perform a division operation and round to the nearest multiple of 7 in the direction of zero.

```
fixDiv(int16(201),10,7)
```

```
ans =
    14

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Divide and Generate Code

Define a function that uses `fixDiv`.

```
function y = fixDiv_example(x,d)
y = fixDiv(x,d);
end
```

Define inputs and execute the function in MATLAB®.

```
x = fi(pi);
d = fi(2);
y = fixDiv_example(x,d)
```

```
y =
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

To generate code for this function, the denominator `d` must be defined as a constant.

```
codegen fixDiv_example -args {x, coder.Constant(d)}
```

```
Code generation successful.
```

Alternatively, you can define the denominator, `d`, as constant in the body of the code.

```
function y = fixDiv10(x)
y = fixDiv(x,10);
end
```

```
x = fi(5*pi);
y = fixDiv10(x)
```

```
y =
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

```
codegen fixDiv10 -args {x}
```

```
Code generation successful.
```

Input Arguments

x — Dividend
scalar

Dividend, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

d — Divisor
scalar

Divisor, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

m — Value to round to nearest multiple of
1 (default) | scalar

Value to round to nearest multiple of, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

Output Arguments

y — Result of division and round to zero
scalar

Result of division and round to zero, returned as a scalar.

The datatype of y is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of x, and the values of d and m.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

To generate code, the denominator d must be declared as constant.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also
ceilDiv | floorDiv | nearestDiv

fixed.aggregateType

Compute aggregate numerictype

Syntax

`aggNT = fixed.aggregateType(A,B)`

Description

`aggNT = fixed.aggregateType(A,B)` computes the smallest binary point scaled numerictype that is able to represent both the full range and precision of inputs `A` and `B`.

Examples

Compute Aggregate Numeric Type

Aggregate Numeric Type of Two `numerictype` Objects

```
a_nt = numerictype(1,16,13);
b_nt = numerictype(1,18,16);
aggNT = fixed.aggregateType(a_nt,b_nt)
```

```
aggNT =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 19
        FractionLength: 16
```

`a_nt` can represent the range `[-4,4)` with precision `2^-13`. `b_nt` can represent the range `[-2,2)` with precision `2^-16`. `aggNT` can represent the range `[-4,4)` with precision `2^-16`.

Aggregate Numeric Type of Two `fi` Objects

```
a_fi = ufi(pi,16)
```

```
a_fi =
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 14
```

```
b_fi = sfi(-pi,24)
```

```
b_fi =
   -3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
           WordLength: 24
       FractionLength: 21
```

```
aggNT = fixed.aggregateType(a_fi,b_fi)
```

```
aggNT =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 21
```

Aggregate Numeric Type of a fi Object and an Integer

```
a_fi = ufi(pi,16);
cInt = uint8(0);
aggNT = fixed.aggregateType(a_fi,cInt)
```

```
aggNT =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 22
        FractionLength: 14
```

`a_fi` can represent the range `[0,3]` with precision `2^-14`. `cInt` can represent the range `[0,255]` with precision `2^0`. `aggNT` can represent the range `[0,255]` with precision `2^-14`.

Input Arguments

A,B — Input objects

integers | `numerictype` objects | `fi` objects

Input objects, specified as integers, binary point scaled fixed-point `fi` objects, or `numerictype` objects.

Output Arguments

aggNT — Aggregate numeric type

`numerictype` object

Aggregate numeric type, returned as a `numerictype` object.

Version History

Introduced in R2011b

See Also

`numerictype` | `fi`

fixed.backwardSubstitute

Solve upper-triangular system of equations through backward substitution

Syntax

```
x = fixed.backwardSubstitute(R, C)
x = fixed.backwardSubstitute(R, C, outputType)
```

Description

`x = fixed.backwardSubstitute(R, C)` performs backward substitution on upper-triangular matrix `R` to compute $x = R \backslash C$.

`x = fixed.backwardSubstitute(R, C, outputType)` returns $x = R \backslash C$, where the data type of output variable, `x`, is specified by `outputType`.

Examples

Solve a System of Equations Using Forward and Backward Substitution

This example shows how to solve the system of equations $(A'A)x = B$ using forward and backward substitution.

Specify the input variables, `A` and `B`.

```
rng default;
A = gallery('randsvd', [5,3], 1000);
b = [1; 1; 1; 1; 1];
```

Compute the upper-triangular factor, `R`, of `A`, where $A = QR$.

```
R = fixed.qlessQR(A);
```

Use forward and backward substitution to compute the value of `X`.

```
X = fixed.forwardSubstitute(R,b);
X(:) = fixed.backwardSubstitute(R,X)
```

```
X = 5×1
10^5 ×

   -0.9088
    2.7123
   -0.8958
         0
         0
```

This solution is equivalent to using the `fixed.qlessQRMatrixSolve` function.

```
x = fixed.qlessQRMatrixSolve(A,b)
```

```
x = 5×1
10^5 ×

   -0.9088
    2.7123
   -0.8958
         0
         0
```

Input Arguments

R — Upper-triangular input matrix
matrix

Upper triangular input, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

C — Linear system factor
matrix

Linear system factor, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

outputType — Output data type
`numerictype` object | numeric variable

Output data type, specified as a `numerictype` object or a numeric variable. If `outputType` is specified as a `numerictype` object, the output, `x`, will have the specified data type. If `outputType` is specified as a numeric variable, `x` will have the same data type as the numeric variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi` | `numerictype`

Output Arguments

x — Solution
matrix

Solution, returned as a matrix satisfying the equation $x = R\backslash C$.

Version History
Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

R and *C* must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.forwardSubstitute` | `fixed.qlessQR` | `fixed.qlessQRUpdate` | `fixed.qrAB` | `fixed.qrMatrixSolve` | `fixed.qlessQRMatrixSolve`

fixed.complexConditionNumberUpperBound

Estimate of upper bound for 2-norm condition number of complex-valued matrix

Syntax

```
C = fixed.complexConditionNumberUpperBound(m,n,max_abs_A,
noiseStandardDeviation)
C = fixed.complexConditionNumberUpperBound( ___ ,p_s)
C = fixed.complexConditionNumberUpperBound( ___ ,regularizationParameter)
```

Description

`C = fixed.complexConditionNumberUpperBound(m,n,max_abs_A, noiseStandardDeviation)` returns an estimate of an upper bound for the 2-norm condition number of a complex-valued `m`-by-`n` matrix `A`, where `max_abs_A >= max(abs(A(:)))` and `noiseStandardDeviation` is the standard deviation of the additive random noise in `A`.

`C = fixed.complexConditionNumberUpperBound( ___ ,p_s)` uses the probability `p_s` that the estimate of the lower bound of the smallest singular value is larger than the actual smallest singular value. `p_s` is an optional parameter. If not supplied or empty, then the default value is used.

`C = fixed.complexConditionNumberUpperBound( ___ ,regularizationParameter)` returns an estimate of an upper bound for the 2-norm condition number of a complex-valued matrix $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$, where λ is the `regularizationParameter`, A is an m-by-n matrix with $m >= n$, and I_n = `eye(`*n*`)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Condition Number of Low Rank Matrix with Additive Noise

Estimate an upper bound for the 2-norm condition number of a low rank, complex-valued matrix with additive noise.

Define a complex matrix `A` with additive noise.

```
m = 300;
n = 10;
rankA = 3;
A = fixed.example.complexRandomLowRankMatrix(m,n,rankA);

noiseStandardDeviation = sqrt(10^(-50/10));
A = A + fixed.example.complexNormalRandomArray(0,...
    noiseStandardDeviation,m,n);
```

Estimate an upper bound for the condition number of the matrix `A`.

```
C = fixed.complexConditionNumberUpperBound(m,n,...
    max(abs(A(:))),noiseStandardDeviation)
```

```
C =

   1.4375e+03
```

Compare to the actual condition number of the matrix.

```
C_actual = cond(A)
```

```
C_actual =

  304.4858
```

Condition Number of Low Rank Matrix with Regularization Parameter

Estimate an upper bound for the 2-norm condition number of a low rank, complex-valued matrix with additive noise, using the regularization parameter.

Define a complex matrix A with additive noise.

```
m = 300;
n = 10;
rankA = 3;
A = fixed.example.complexRandomLowRankMatrix(m,n,rankA);

noiseStandardDeviation = sqrt(10^(-50/10));
A = A + fixed.example.complexNormalRandomArray(0,...
    noiseStandardDeviation,m,n);
```

Define the regularization parameter.

```
regularizationParameter = 0.01;
A = [regularizationParameter*eye(n);A];
```

Estimate an upper bound for the condition number of the matrix A with the regularization parameter. Use the default value for p_s.

```
C = fixed.complexConditionNumberUpperBound(m,n,max(abs(A(:))),...
    noiseStandardDeviation,[],regularizationParameter)
```

```
C =

   1.3946e+03
```

Compare to the actual condition number of the matrix.

```
C_actual = cond(A)
```

```
C_actual =

  291.7264
```

Condition Number of Full Rank Matrix

Estimate an upper bound for the 2-norm condition number of a full rank random matrix with normally distributed elements.

Define a full rank, random, complex matrix A with normally distributed elements.

```
m = 300;
n = 10;
noiseStandardDeviation = 1;
A = fixed.example.complexNormalRandomArray(0,...
    noiseStandardDeviation,m,n);
```

Estimate an upper bound for the condition number of the matrix A.

```
C = fixed.complexConditionNumberUpperBound(m,n,...
    max(abs(A(:))),noiseStandardDeviation)
```

```
C =

   12.0438
```

Compare to the actual condition number of the matrix.

```
C_actual = cond(A)
```

```
C_actual =

    1.3560
```

Input Arguments

m — Number of rows in matrix A
positive integer-valued scalar

Number of rows in matrix A, specified as a positive integer-valued scalar. The number of rows, m, must be greater than or equal to the number of columns, n.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

n — Number of columns in matrix A
positive integer-valued scalar

Number of columns in matrix A, specified as a positive integer-valued scalar. The number of rows, m, must be greater than or equal to the number of columns, n.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

max_abs_A — Maximum of absolute value of matrix A
scalar

Maximum of absolute value of matrix A, specified as a scalar.

Example: max_abs_A >= max(abs(A(:)))

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

noiseStandardDeviation — Standard deviation of additive random noise in matrix A
(2^-precisionBits)/(sqrt(12)) (default) | scalar

Standard deviation of additive random noise in matrix A, specified as a scalar.

If noiseStandardDeviation is not supplied or empty, then the default value is used, which is the standard deviation of the quantization noise,

$$\sigma_q = \frac{2^{-precisionBits}}{\sqrt{12}}.$$

This value is calculated by the function `fixed.complexQuantizationNoiseStandardDeviation`.

If `noiseStandardDeviation` is zero, then `fixed.singularValueLowerBound` will return zero for the estimate of the smallest singular value and `fixed.complexConditionNumberUpperBound` will return an infinite condition number.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

p_s — Probability that estimate of lower bound of smallest singular value is larger than actual smallest singular value of matrix A
`2.8665e-07` (default) | scalar

Probability that estimate of lower bound of smallest singular value is larger than actual smallest singular value of matrix `A`, specified as a scalar.

If `p_s` is not supplied or empty, then the default of `p_s = (1/2)*(1+erf(-5/sqrt(2))) = 2.8665e-07` is used, which is five standard deviations below the mean. So, the probability that the estimated lower bound for the smallest singular value is less than the actual smallest singular value is `1 - p_s = 0.99999971 - p_s = 0.9999997`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar.

`regularizationParameter` is the Tikhonov regularization parameter of the matrix `[regularizationParameter*eye(n); A]`, where `A` is an `m`-by-`n` matrix with `m >= n`.

Data Types: `single` | `double`

More About

Condition Number for Inversion

A *condition number* for a matrix and computational task measures how sensitive the answer is to changes in the input data and roundoff errors in the solution process. The *condition number for inversion* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. The condition number for inversion gives an indication of the accuracy of the results from matrix inversion and the linear equation solution.

A large condition number indicates that a small change in the coefficient matrix `A` can lead to larger changes in the output b in the linear equation $Ax = b$. The extreme case is when `A` is so poorly conditioned that it is singular (an infinite condition number), in which case it has no inverse and the linear equation has no unique solution.

Algorithms

The condition number with respect to the inversion of matrix `A` is the ratio of the largest singular value of `A` to the smallest singular value of `A`. The `fixed.complexSingularValueLowerBound` function estimates the lower bound of the smallest singular value, `s_n`, of `A`. The `fixed.singularValueUpperBound` function determines an upper bound for the largest singular value, `svdUpperBound`, of `A`. A bound on the condition number of `A` is then `cond(A) = max(svd(A))/min(svd(A)) <= svdUpperBound/s_n` [1][2][3].

Version History

Introduced in R2022b

References

[1] Bryan, Thomas A., Jenna L. Warren, Brenda Zhuang, and Jessica Clayton. Continuation in Part for "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2022.

[2] Bryan, Thomas A. and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[3] Chen, Zizhong and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no. 3 (July 2005): 603-620. https://doi.org/10.1137/040616413.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fixed.singularValueUpperBound` | `fixed.complexSingularValueLowerBound` | `fixed.realConditionNumberUpperBound` | `fixed.complexQuantizationNoiseStandardDeviation` | `cond`

fixed.complexQlessQRMatrixSolveFixedpointTypes

Determine fixed-point types for matrix solution of complex-valued $A'AX=B$ using QR decomposition

Syntax

```
T = fixed.complexQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,
precisionBits)
T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,
noiseStandardDeviation)
T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,p_s)
T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,
regularizationParameter)
T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)
```

Description

`T = fixed.complexQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,precisionBits)` computes fixed-point types for the matrix solution of complex-valued $A'AX=B$ using QR decomposition. T is returned as a struct with fields that specify fixed-point types for A and B that guarantee no overflow will occur in the QR algorithm transforming A in-place into upper-triangular R, where $QR=A$ is the QR decomposition of X, and X such that there is a low probability of overflow.

`T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)` specifies the standard deviation of the additive random noise in A. `noiseStandardDeviation` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,p_s)` specifies the probability that the estimate of the lower bound for the smallest singular value of A is larger than the actual smallest singular value of the matrix. `p_s` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)` computes fixed-point types for the matrix solution of complex-valued

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the `regularizationParameter`, A is an m-by-n matrix, and I_n = `eye(n)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.complexQlessQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)` specifies the maximum word length of the fixed-point types. `maxWordLenth` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Determine Fixed-Point Types for Complex Q-less QR Matrix Solve A'AX=B

This example shows how to use the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the complex least-squares matrix equation $A'AX = B$, where A is an m-by-n matrix with $m \geq n$, B is n-by-p, and X is n-by-p.

Fixed-point types for the solution of the matrix equation $A'AX = B$ are well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices B and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\text{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, complex-valued matrices `A` and `B` are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of −50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use `fixed.complexQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBit
```

```
quantizationNoiseStandardDeviation = 2.4333e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function to compute fixed-point types.

```
T = fixed.complexQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to $R = Q'A$ in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans =

[]
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 27
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = (A'A)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 24
```

Use the Specified Types to Solve the Matrix Equation A'AX=B

Create random matrices `A` and `B` such that `rankA=rank(A)`. Add random measurement noise to `A` which will make it become full rank.

```
rng('default');
[A,B] = fixed.example.complexRandomQlessQRMatrices(m,n,p,rankA);
A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.complexQlessQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise.

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qlessQRMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qlessQRMatrixSolve -args {A,B,T.X} -o qlessQRMatrixSolve_mex
```

Specify output type `T.X` and compute fixed-point $X = (A'A)\backslash B$ using the QR method.

```
X = qlessQRMatrixSolve_mex(A,B,T.X);
```

Compute the relative error to verify the accuracy of the output.

```
relative_error = norm(double(A'*A*X - B))/norm(double(B))
```

```
relative_error = 0.1054
```

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Complex Q-less QR Matrix Solve with Tikhonov Regularization

This example shows how to use the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the complex least-squares matrix equation

$$\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}^{\mathrm{H}}\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}X = (\lambda^2 I_n + A^{\mathrm{H}}A)X = B$$

where A is an m-by-n matrix with $m \geq n$, B is n-by-p, X is n-by-p, $I_n = \mathrm{eye}(n)$, and λ is a regularization parameter.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices B and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\mathrm{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 32;
```

Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

```
regularizationParameter = 0.01;
```

In this example, complex-valued matrices `A` and `B` are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use `fixed.complexQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBi
```

```
quantizationNoiseStandardDeviation = 9.5053e-11
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function to compute fixed-point types.

```
T = fixed.complexQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation,[],regularizationParameter)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ to $R = Q^{\mathrm{H}}\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
        WordLength: 40
    FractionLength: 32
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans =

[]

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 35
    FractionLength: 32
```

`T.X` is the type computed for the solution $X = \left(\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}^{\mathrm{H}}\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}\right)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 48
    FractionLength: 32
```

Use the Specified Types to Solve the Matrix Equation

Create random matrices `A` and `B` such that `rankA=rank(A)`. Add random measurement noise to `A` which will make it become full rank.

```
rng('default');
[A,B] = fixed.example.complexRandomQlessQRMatrices(m,n,p,rankA);
A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.complexQlessQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise.

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qlessQRMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel +fixed/qlessQRMatrixSolve -args {A,B,T.X,[],regularizationParameter} -o qlessQRMatrixSol
```

Specify output type `T.X` and compute fixed-point $X = \left(\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}^{\mathrm{H}}\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}\right)\backslash B$ using the QR method.

```
X = qlessQRMatrixSolve_mex(A,B,T.X,[],regularizationParameter);
```

Verify the Accuracy of the Output

Verify that the relative error between the fixed-point output and builtin MATLAB in double-precision floating-point is small.

$$X_{\text{double}} = \left(\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}^{\mathrm{H}} \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} \right) \backslash B$$

```
A_lambda = double([regularizationParameter*eye(n);A]);
X_double = (A_lambda'*A_lambda)\double(B);
relativeError = norm(X_double - double(X))/norm(X_double)
```

```
relativeError = 1.0591e-05
```

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Input Arguments

`m` — Number of rows in *A* and *B*
positive integer-valued scalar

Number of rows in A and B, specified as a positive integer-valued scalar.

Data Types: `double`

`n` — Number of columns in *A*
positive integer-valued scalar

Number of columns in A, specified as a positive integer-valued scalar.

Data Types: `double`

`max_abs_A` — Maximum of absolute value of *A*
scalar

Maximum of the absolute value of A, specified as a scalar.

Example: `max(abs(A(:)))`

Data Types: `double`

`max_abs_B` — Maximum of absolute value of *B*
scalar

Maximum of the absolute value of B, specified as a scalar.

Example: `max(abs(B(:)))`

Data Types: `double`

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision of the input and output, specified as a positive integer-valued scalar.

Data Types: double

noiseStandardDeviation — Standard deviation of additive random noise in *A*
scalar

Standard deviation of additive random noise in A, specified as a scalar.

If noiseStandardDeviation is not specified, then the default is the standard deviation of the complex-valued quantization noise $\sigma_q = \left(2^{-\text{precisionBits}}\right)/(\sqrt{6})$, which is calculated by fixed.complexQuantizationNoiseStandardDeviation.

Data Types: double

p_s — Probability that estimate of lower bound *s* is larger than actual smallest singular value of matrix
≈3·10-7 (default) | scalar

Probability that estimate of lower bound s is larger than actual smallest singular value of matrix, specified as a scalar. Use fixed.complexSingularValueLowerBound to estimate the smallest singular value, s, of A. If p_s is not specified, the default value is $p_s = (1/2) \cdot (1 + \text{erf}(-5/\sqrt{2})) \approx 3 \cdot 10^{-7}$ which is 5 standard deviations below the mean, so the probability that the estimated bound for the smallest singular value is less than the actual smallest singular value is 1-$p_s \approx 0.9999997$.

Data Types: double

regularizationParameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

regularizationParameter is the Tikhonov regularization parameter of the matrix problem

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

maxWordLength — Maximum word length of fixed-point types
128 (default) | positive integer

Maximum word length of fixed-point types, specified as a positive integer.

If the word length of the fixed-point type exceeds the specified maximum word length, the default of 128 bits is used.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

Output Arguments

T — Fixed-point types for *A*, *B*, and *X*
struct

Fixed-point types for A, B, and X, returned as a struct. The struct `T` has fields `T.A`, `T.B`, and `T.X`. These fields contain `fi` objects that specify fixed-point types for

- A and B that guarantee no overflow will occur in the QR algorithm.

  The QR algorithm transforms A in-place into upper-triangular R where $QR=A$ is the QR decomposition of A.
- X such that there is a low probability of overflow.

Tips

Use `fixed.complexQlessQRMatrixSolveFixedpointTypes` to compute fixed-point types for the inputs of these functions and blocks.

- `fixed.qlessQRMatrixSolve`
- Complex Burst Matrix Solve Using Q-less QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Algorithms

The fixed-point type for A is computed using `fixed.qlessqrFixedpointTypes`. The required number of integer bits to prevent overflow is derived from the following bound on the growth of R [1]. The required number of integer bits is added to the number of bits of precision, `precisionBits`, of the input, plus one for the sign bit, plus one bit for intermediate CORDIC gain of approximately 1.6468 [2].

The elements of R are bounded in magnitude by

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

Matrix B is not transformed, so it does not need any additional growth bits.

The elements of $X=R\backslash(R'\backslash B)$ are bounded in magnitude by

$$\max(|X(:)|) \leq \frac{n \cdot \max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Computing the singular value decomposition to derive the above bound on X is more computationally intensive than the entire matrix solve, so the `fixed.complexSingularValueLowerBound` function is used to estimate a bound on `min(svd(A))`.

Version History

Introduced in R2021b

R2022b: Support for maximum word length

You can now use the `maxWordLenth` parameter to specify the maximum word length of the fixed-point types.

R2022a: Support for Tikhonov regularization parameter

The `fixed.complexQlessQRMatrixSolveFixedpointTypes` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

References

[1] “Perform QR Factorization Using CORDIC”

[2] Voler, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers* EC-8 (1959): 330-334.

See Also

Functions
`fixed.complexQuantizationNoiseStandardDeviation` | `fixed.complexSingularValueLowerBound` | `fixed.qlessqrFixedpointTypes` | `fixed.qlessQRMatrixSolve`

Blocks
Complex Burst Matrix Solve Using Q-less QR Decomposition | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

fixed.complexQRMatrixSolveFixedpointTypes

Determine fixed-point types for matrix solution of complex-valued $AX=B$ using QR decomposition

Syntax

```
T = fixed.complexQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,
precisionBits)
T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)
T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,p_s)
T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)
T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)
```

Description

`T = fixed.complexQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B, precisionBits)` computes fixed-point types for the matrix solution of complex-valued $AX=B$ using QR decomposition. T is returned as a struct with fields that specify fixed-point types for A and B that guarantee no overflow will occur in the QR algorithm, and X such that there is a low probability of overflow.

The QR algorithm transforms A in-place into upper-triangular R and transforms B in-place into $C=Q'B$, where $QR=A$ is the QR decomposition of A.

`T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)` specifies the standard deviation of the additive random noise in A. `noiseStandardDeviation` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,p_s)` specifies the probability that the estimate of the lower bound for the smallest singular value of A is larger than the actual smallest singular value of the matrix. `p_s` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)` computes fixed-point types for the matrix solution of complex-valued $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix} X = \begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$ where λ is the `regularizationParameter`, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(`*n*`)`, and $0_{n,p}$ = `zeros(`*n*`,`*p*`)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.complexQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)` specifies the maximum word length of the fixed-point types. `maxWordLength` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Algorithms to Determine Fixed-Point Types for Complex Q-less QR Matrix Solve A'AX=B

This example shows the algorithms that the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the complex matrix equation $A'AX = B$, where A is an m-by-n matrix with $m \geq n$, B is n-by-p, and X is n-by-p.

Overview

You can solve the fixed-point matrix equation $A'AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $R'RX = B$. To solve for X, compute $X = R\backslash(R'\backslash B)$ through forward- and backward-substitution of R into B.

You can determine appropriate fixed-point types for the matrix equation $A'AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.complexQlessQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on R, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of $R = Q'A$ is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \leq \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Since computing svd(A) is more computationally expensive than solving the system of equations, the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function estimates a lower bound of min(svd(A)).

Fixed-point types for the solution of the matrix equation $(A'A)X = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a^2/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned} ||Av||_2 &\leq ||A||_2||v||_2 \\ ||Q||_2 &= 1 \\ ||v||_\infty &= \max(|v(:)|) \\ ||v||_\infty &\leq ||v||_2 \leq \sqrt{m}||v||_\infty \end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$.

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:,j) = Q'A(:,j)$, so

$$\begin{aligned}
\max(|R(:,j)|) &= ||R(:,j)||_\infty \\
&\le ||R(:,j)||_2 \\
&= ||Q'A(:,j)||_2 \\
&\le ||Q'||_2\,||A(:,j)||_2 \\
&= ||A(:,j)||_2 \\
&\le \sqrt{m}\,||A(:,j)||_\infty \\
&= \sqrt{m}\max(|A(:,j)|) \\
&\le \sqrt{m}\max(|A(:)|)\,.
\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)\,.$$

Upper Bound for X = (A'A)\B

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$\max(|X(:)|) \le \dfrac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}$.

Proof of Upper Bound for X = (A'A)\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{n}\max(|B(:)|)/\min(\mathrm{svd}(A))^2 = \infty$ and so the inequality is true.

If $A'Ax = b$ and $QR = A$ is the economy-size QR decomposition of A, then $A'Ax = R'Q'QRx = R'Rx = b$. If A is full rank then $x = R^{-1} \cdot ((R')^{-1}b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the jth column of B. Then

$$\begin{aligned}
\max(|x(:)|) &= ||x||_\infty \\
&\le ||x||_2 \\
&= ||R^{-1}\cdot((R')^{-1}b)||_2 \\
&\le ||R^{-1}||_2\,||(R')^{-1}||_2\,||b||_2 \\
&= \left(1/\min(\mathrm{svd}(A))^2\right)\cdot ||b||_2 \\
&= ||b||_2/\min(\mathrm{svd}(A))^2 \\
&\le \sqrt{n}\,||b||_\infty/\min(\mathrm{svd}(A))^2 \\
&= \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2 .
\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\mathrm{svd}(A))$for complex-valued A using the following formula,

$$s = \frac{\sigma_N}{\sqrt{2}}\sqrt{\gamma^{-1}\left(\frac{p_s\,\Gamma(m-n+2)^2\,\Gamma(n)}{\Gamma(m+1)\,\Gamma(m-n+1)(m-n+1)},\ m-n+1\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1-p_s$ is the probability that $s \le \min(\mathrm{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.4 from [3] and rearranging terms.

Since $s \le \min(\mathrm{svd}(A))$ with probability $1-p_s$, then you can bound the magnitude of the elements of X without computing $\mathrm{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2} \le \frac{\sqrt{n}\max(|B(:)|)}{s^2} \text{ with probability } 1-p_s.$$

You can compute s using the `fixed.complexSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean, $p_s = (1+\mathrm{erf}(-5/\sqrt{2}))/2 \approx 2.8665\cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1-p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, and $X = (A'A)\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices `B` and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\text{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, complex-valued matrices `A` and `B` are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing the real and imaginary parts of a complex signal is $2^{-\text{precisionBits}}/\sqrt{6}$ [4,5]. Use `fixed.complexQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBi
```

```
quantizationNoiseStandardDeviation = 2.4333e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.complexQlessQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.complexQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 27
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = (A'A)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
          WordLength: 40
      FractionLength: 24
```

Upper Bound for R

The upper bound for R is computed using the formula $\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$, where m is the number of rows of matrix A. This upper bound is used to select a fixed-point type with the required number of bits of precision to avoid an overflow in the upper bound.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 24.4949
```

Lower Bound for min(svd(A)) for Complex A

A lower bound for min(svd(A)) is estimated by the `fixed.complexSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.complexSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviatio
```

```
estimatedSingularValueLowerBound = 0.0389
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,numSamples,
    noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 24.4949
```

```
max(actualMaxR)
```

```
ans = 9.4990
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

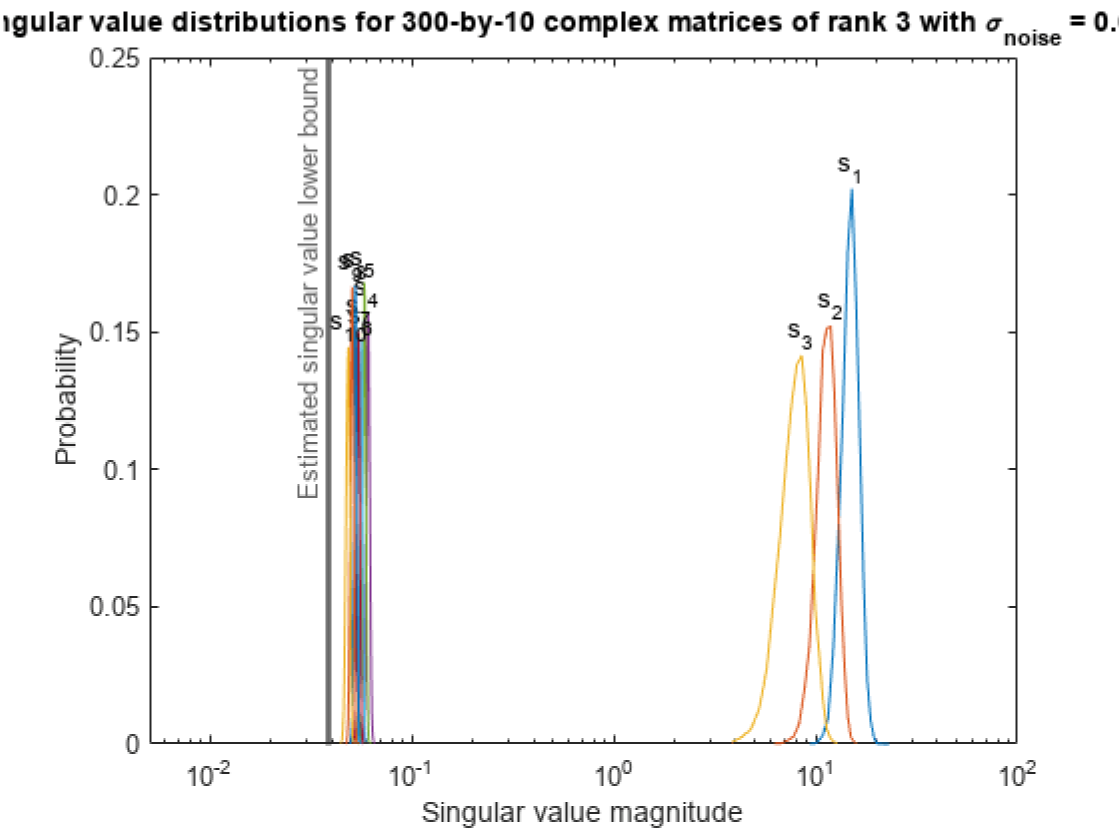
```
estimatedSingularValueLowerBound = 0.0389
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0443
```

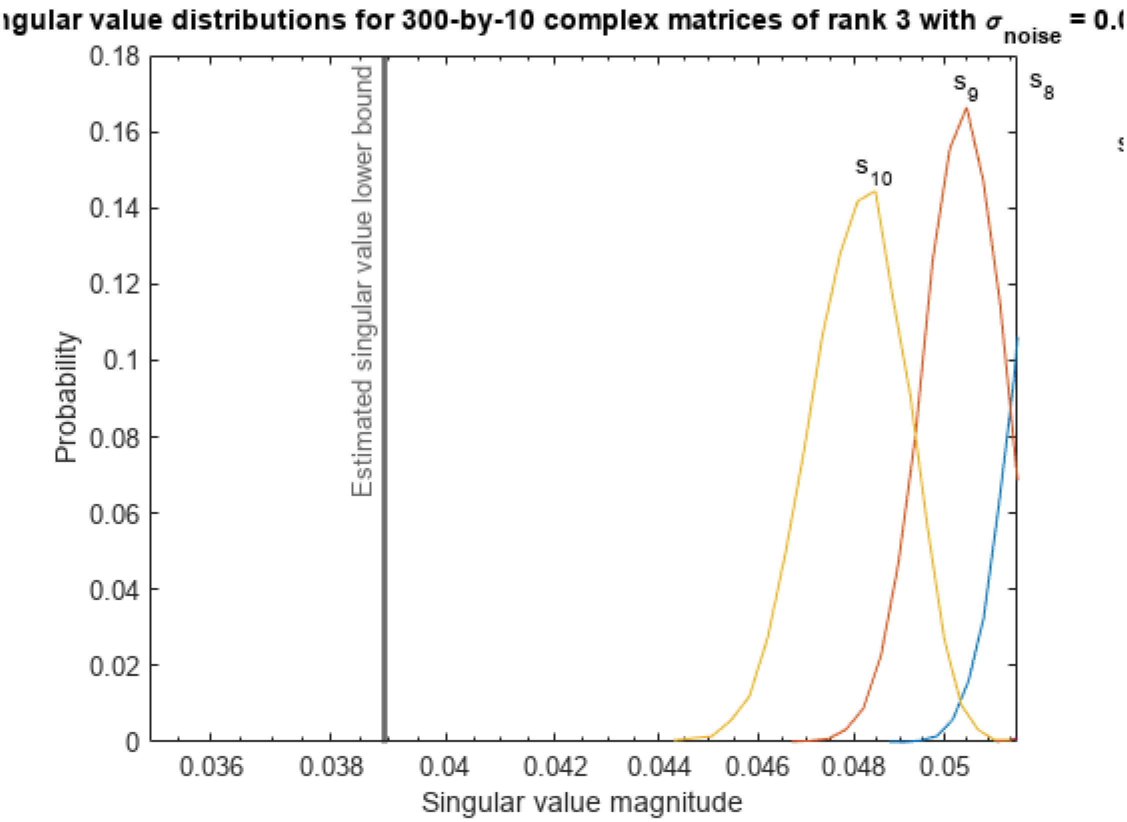
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,...
    noiseStandardDeviation,singularValues,...
    estimatedSingularValueLowerBound,"complex");
```



Zoom in to the smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.complexQlessQRMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDevia
```

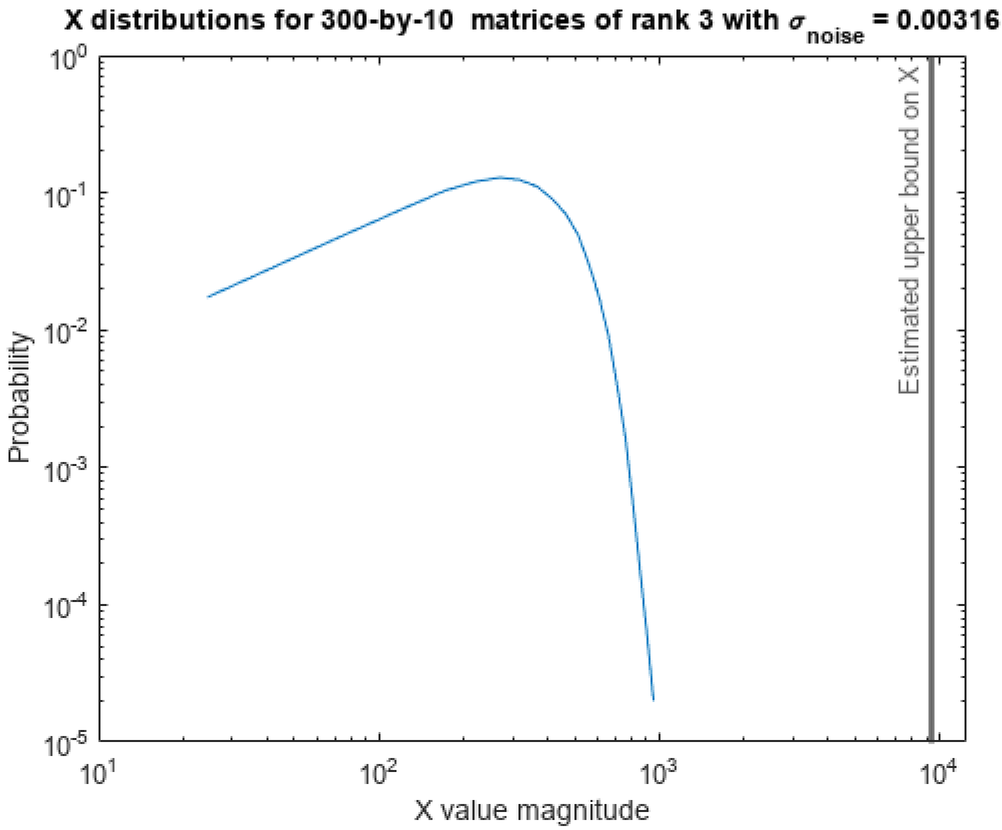
```
estimated_largest_X = 9.3348e+03
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 977.7440
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"complex normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $A'AX = B$. It returns the maximum values of $R = Q'A$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,...
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = (max_abs_A/sqrt(2))*fixed.example.complexRandomLowRankMatrix(m,n,rankA);
        % Adding random noise makes A non-singular.
        A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.complexUniformRandomArray(-max_abs_B,max_abs_B,n,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [~,R] = qr(A,0);
        X = R\(R'\B);
        actualMaxR(j) = max(abs(R(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
```

```
    end
end
```

References

1 Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.

2 “Perform QR Factorization Using CORDIC”. Derivation of the bound on growth when computing QR. MathWorks. 2010.

3 Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.

4 Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

5 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

6 Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Algorithms to Determine Fixed-Point Types for Complex Least-Squares Matrix Solve AX=B

This example shows the algorithms that the `fixed.complexQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the complex least-squares matrix equation $AX = B$, where A is an m-by-n matrix with $m \geq n$, B is m-by-p, and X is n-by-p.

Overview

You can solve the fixed-point least-squares matrix equation $AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, and transforms matrix B in-place to $C = Q'B$, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $RX = C$. To solve for X, compute $X = R\backslash C$ through back-substitution of R into C.

You can determine appropriate fixed-point types for the least-squares matrix equation $AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.complexQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on $R = Q'A$, $C = Q'B$, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of $R = Q'A$ is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

The upper bound for the magnitude of the elements of $C = Q'B$ is

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|).$$

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$$\max(|X(:)|) \leq \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}.$$

Since computing svd(A) is more computationally expensive than solving the system of equations, the `fixed.complexQRMatrixSolveFixedpointTypes` function estimates a lower bound of min(svd(A)).

Fixed-point types for the solution of the matrix equation $AX = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned}
||Av||_2 &\leq ||A||_2||v||_2 \\
||Q||_2 &= 1 \\
||v||_\infty &= \max(|v(:)|) \\
||v||_\infty &\leq ||v||_2 \leq \sqrt{m}||v||_\infty
\end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:, j) = Q'A(:, j)$, so

$$\begin{aligned}\max(|R(:,j)|) &= ||R(:,j)||_\infty \\ &\le ||R(:,j)||_2 \\ &= ||Q'A(:,j)||_2 \\ &\le ||Q'||_2\,||A(:,j)||_2 \\ &= ||A(:,j)||_2 \\ &\le \sqrt{m}\,||A(:,j)||_\infty \\ &= \sqrt{m}\max(|A(:,j)|) \\ &\le \sqrt{m}\max(|A(:)|)\,.\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)\,.$$

Upper Bound for C = Q'B

The upper bound for the magnitude of the elements of $C = Q'B$ is

$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|)$.

Proof of Upper Bound for C = Q'B

The proof of the upper bound for $C = Q'B$ is the same as the proof of the upper bound for $R = Q'A$ by substituting C for R and B for A.

Upper Bound for X = A\B

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$\max(|X(:)|) \le \dfrac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}$.

Proof of Upper Bound for X = A\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{m}\max(|B(:)|)/\min(\mathrm{svd}(A)) = \infty$ and so the inequality is true.

If A is full rank, then $x = R^{-1}(Q'b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the jth column of B. Then

$$\begin{aligned}\max(|x(:)|) &= ||x||_\infty \\ &\le ||x||_2 \\ &= ||R^{-1}\cdot(Q'b)||_2 \\ &\le ||R^{-1}||_2\,||Q'||_2\,||b||_2 \\ &= (1/\min(\mathrm{svd}(A)))\cdot 1\cdot ||b||_2 \\ &= ||b||_2/\min(\mathrm{svd}(A)) \\ &\le \sqrt{m}\,||b||_\infty/\min(\mathrm{svd}(A)) \\ &= \sqrt{m}\max(|b(:)|)/\min(\mathrm{svd}(A))\,.\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{m}\max(|b(:)|)/\min(\text{svd}(A))$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\text{svd}(A))$for complex-valued A using the following formula,

$$s = \frac{\sigma_N}{\sqrt{2}}\sqrt{\gamma^{-1}\left(\frac{p_s\Gamma(m-n+2)^2\Gamma(n)}{\Gamma(m+1)\Gamma(m-n+1)(m-n+1)},\ m-n+1\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1 - p_s$ is the probability that $s \le \min(\text{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.4 from [3] and rearranging terms.

Since $s \le \min(\text{svd}(A))$ with probability $1 - p_s$, then you can bound the magnitude of the elements of X without computing $\text{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))} \le \frac{\sqrt{m}\max(|B(:)|)}{s} \text{ with probability } 1 - p_s.$$

You can compute s using the `fixed.complexSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean, $p_s = (1 + \text{erf}(-5/\sqrt{2}))/2 \approx 2.8665 \cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1 - p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, $C = Q'B$, and $X = A\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrix `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, complex-valued matrices A and B are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of A and B.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing the real and imaginary parts of a complex signal is $2^{-\text{precisionBits}}/\sqrt{6}$ [4,5]. Use the `fixed.complexQuantizationNoiseStandardDeviation` function to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBi
```

```
quantizationNoiseStandardDeviation = 2.4333e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.complexQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.complexQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.B` is the type computed for transforming B to $Q'B$ in-place so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = A\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 37
        FractionLength: 24
```

Upper Bounds for R and C=Q'B

The upper bounds for R and $C = Q'B$ are computed using the following formulas, where m is the number of rows of matrices A and B.

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|)$$

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|)$$

These upper bounds are used to select a fixed-point type with the required number of bits of precision to avoid overflows.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 24.4949
```

```
upperBoundQB = sqrt(m)*max_abs_B
```

```
upperBoundQB = 24.4949
```

Lower Bound for min(svd(A)) for Complex A

A lower bound for min(svd(A)) is estimated by the `fixed.complexSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.complexSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviatio
```

```
estimatedSingularValueLowerBound = 0.0389
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B
    numSamples,noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 24.4949
```

```
max(actualMaxR)
```

```
ans = 9.6720
```

You can see that the upper bound on $C = Q'B$ compared to the measured simulation results of the maximum value of $C = Q'B$ over all runs is also within an order of magnitude.

```
upperBoundQB
```

```
upperBoundQB = 24.4949
```

```
max(actualMaxQB)
```

```
ans = 4.4764
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

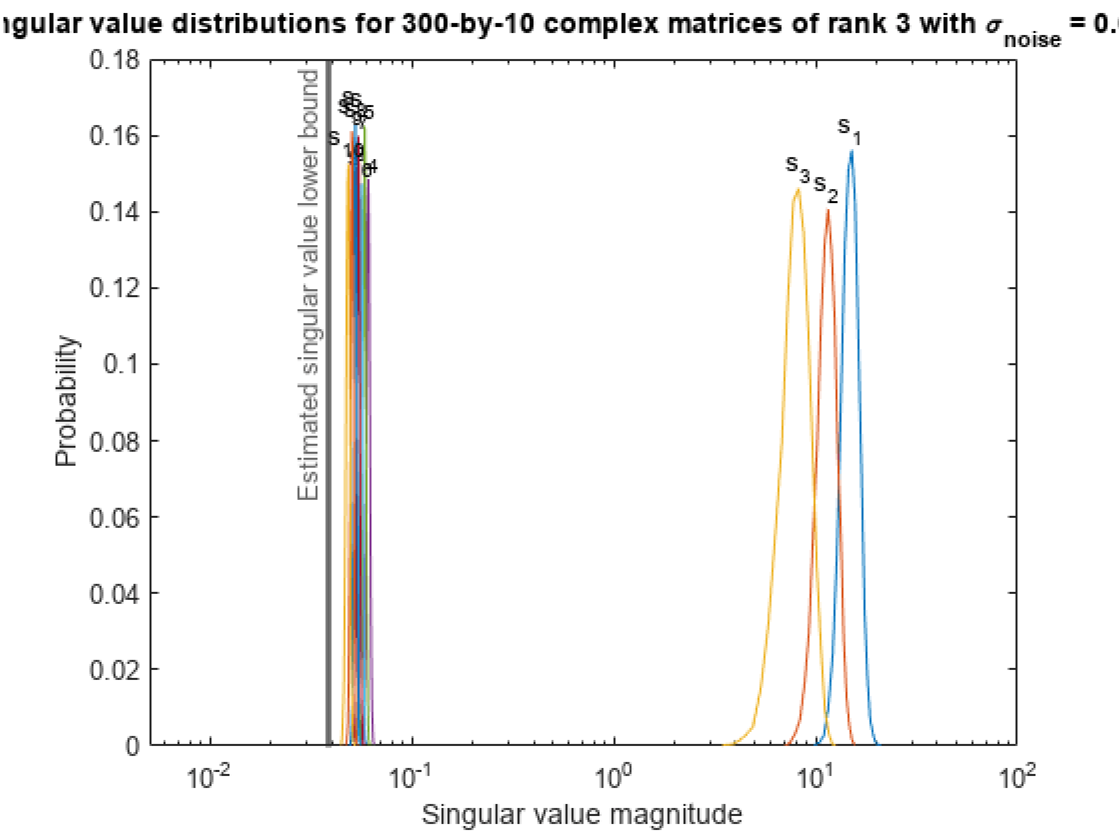
```
estimatedSingularValueLowerBound = 0.0389
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0443
```

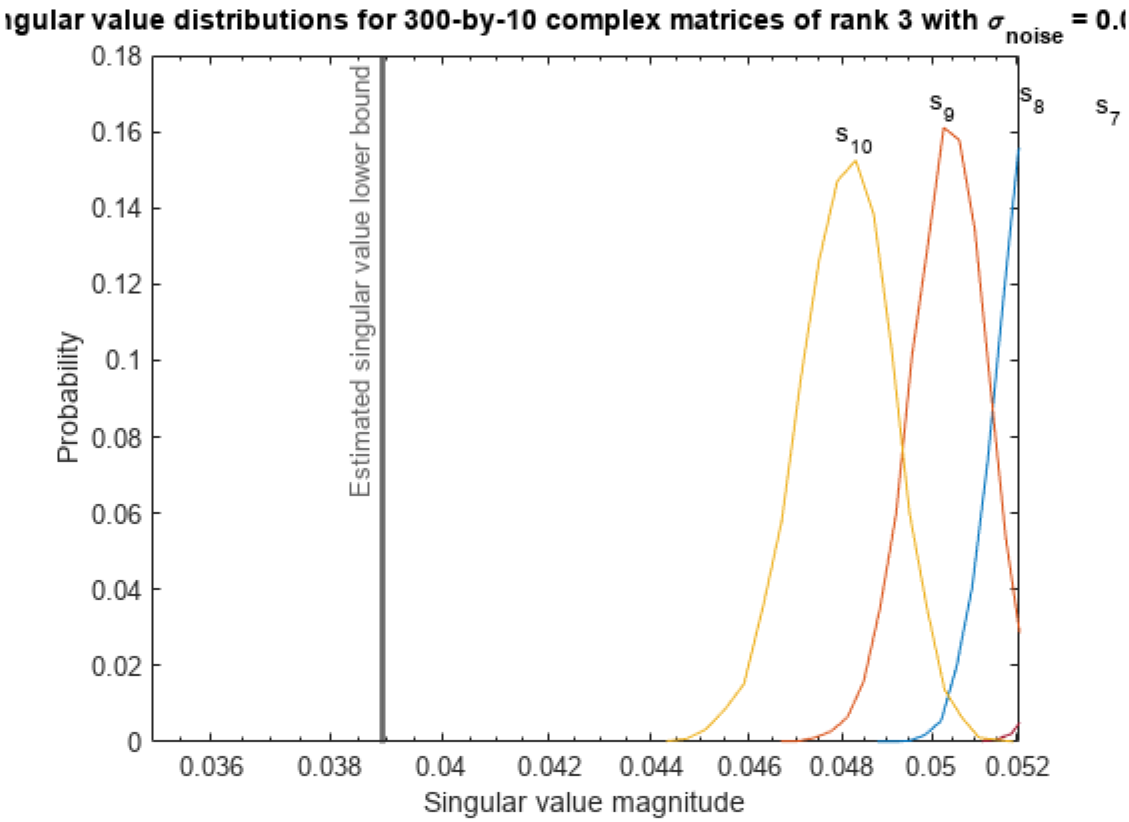
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,noiseStandardDeviation,...
    singularValues,estimatedSingularValueLowerBound,"complex");
```



Zoom in to the smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.complexMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDeviation)
```

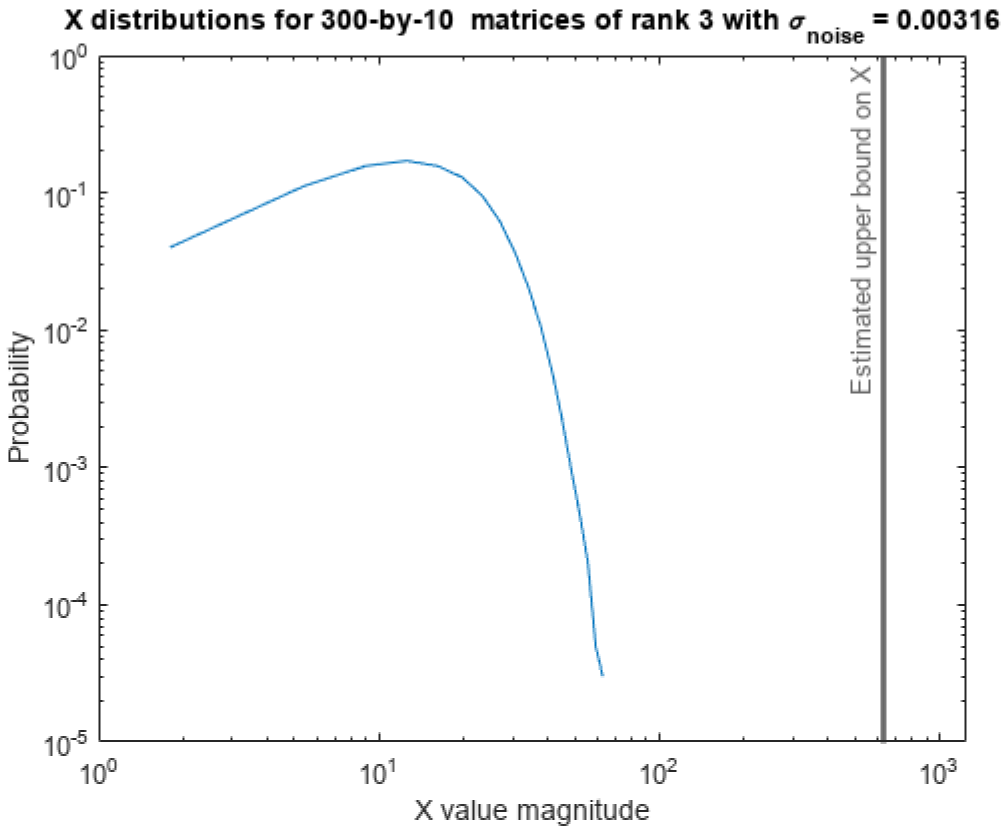
```
estimated_largest_X = 629.3194
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 70.2644
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"complex normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $AX = B$. It returns the maximum values of $R = Q'A$ and $C = Q'B$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    actualMaxQB = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = (max_abs_A/sqrt(2))*fixed.example.complexRandomLowRankMatrix(m,n,rankA);
        % Adding normally distributed random noise makes A non-singular.
        A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.complexUniformRandomArray(-max_abs_B,max_abs_B,m,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [Q,R] = qr(A,0);
        C = Q'*B;
        X = R\C;
        actualMaxR(j) = max(abs(R(:)));
```

```
        actualMaxQB(j) = max(abs(C(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
    end
end
```

References

1. Thomas A. Bryan and Jenna L. Warren. "Systems and Methods for Design Parameter Selection". Patent pending. U.S. Patent Application No. 16/947,130. 2020.
2. Perform QR Factorization Using CORDIC. Derivation of the bound on growth when computing QR. MathWorks. 2010.
3. Zizhong Chen and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices". In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.
4. Bernard Widrow. "A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory". In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.
5. Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.
6. Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Complex Least-Squares Matrix Solve AX=B

This example shows how to use the `fixed.complexQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the complex least-squares matrix equation $AX = B$, where A is an m-by-n matrix with $m \geq n$, B is m-by-p, and X is n-by-p.

Fixed-point types for the solution of the matrix equation $AX = B$ are well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

n is the number of columns in matrix A and rows in matrix X. In a least-squares problem, m is greater than n, and usually m is much larger than n. In a problem such as beamforming or direction finding, n corresponds to the number of sensors.

```
n = 10;
```

p is the number of columns in matrices B and X. It corresponds to simultaneously solving a system with p right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

precisionBits defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, complex-valued matrices A and B are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the upperbound function to determine the upper bounds of the fixed-point types of A and B.

max_abs_A is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

max_abs_B is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set thermalNoiseStandardDeviation to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use fixed.complexQuantizationNoiseStandardDeviation to compute this. See that it is less than thermalNoiseStandardDeviation.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBi
```

```
quantizationNoiseStandardDeviation = 2.4333e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.complexQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.complexQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to $R = Q'A$ in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.B` is the type computed for transforming B to $C = Q'B$ in-place so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = A\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 37
        FractionLength: 24
```

Use the Specified Types to Solve the Matrix Equation AX=B

Create random matrices `A` and `B` such that `B` is in the range of `A`, and `rankA=rank(A)`. Add random measurement noise to `A` which will make it become full rank, but it will also affect the solution so that `B` is only close to the range of `A`.

```
rng('default');
[A,B] = fixed.example.complexRandomLeastSquaresMatrices(m,n,p,rankA);
A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.complexQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise.

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qrMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qrMatrixSolve -args {A,B,T.X} -o qrComplexMatrixSolve_mex
```

Specify the output type `T.X` and compute fixed-point $X = A\backslash B$ using the QR method.

```
X = qrComplexMatrixSolve_mex(A,B,T.X);
```

Compute the relative error to verify the accuracy of the output.

```
relative_error = norm(double(A*X - B))/norm(double(B))
```

```
relative_error = 0.0056
```

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Complex Least-Squares Matrix Solve with Tikhonov Regularization

This example shows how to use the `fixed.complexQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the complex least-squares matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \begin{bmatrix} 0_{n,p} \\ B \end{bmatrix},$$

where A is an m-by-n matrix with $m \geq n$, B is m-by-p, X is n-by-p, $I_n = \text{eye}(n)$, $0_{n,p} = \text{zeros}(n,p)$, and λ is a regularization parameter.

The least-squares solution is

$$X_{LS} = (\lambda^2 I_n + A^{\mathrm{T}} A)^{-1} A^{\mathrm{T}} B$$

but is computed without squares or inverses.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

n is the number of columns in matrix A and rows in matrix X. In a least-squares problem, m is greater than n, and usually m is much larger than n. In a problem such as beamforming or direction finding, n corresponds to the number of sensors.

```
n = 10;
```

p is the number of columns in matrices B and X. It corresponds to simultaneously solving a system with p right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

precisionBits defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 32;
```

Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

```
regularizationParameter = 0.01;
```

In this example, complex-valued matrices A and B are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the upperbound function to determine the upper bounds of the fixed-point types of A and B.

max_abs_A is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

max_abs_B is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set thermalNoiseStandardDeviation to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use fixed.complexQuantizationNoiseStandardDeviation to compute this. See that it is less than thermalNoiseStandardDeviation.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBi
```

```
quantizationNoiseStandardDeviation = 9.5053e-11
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.complexQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.complexQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation,[],regularizationParameter)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ to $R = Q^{\mathrm{T}}\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 32
```

`T.B` is the type computed for transforming $\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ to $C = Q^{\mathrm{T}}\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$ in-place so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 32
```

`T.X` is the type computed for the solution $X = \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} \backslash \begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$, so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 44
        FractionLength: 32
```

Use the Specified Types to Solve the Matrix Equation

Create random matrices A and B such that B is in the range of A, and rankA=rank(A). Add random measurement noise to A which will make it become full rank, but it will also affect the solution so that B is only close to the range of A.

```
rng('default');
[A,B] = fixed.example.complexRandomLeastSquaresMatrices(m,n,p,rankA);
A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by fixed.complexQRMatrixSolveFixedpointTypes. Quantizing to fixed-point is equivalent to adding random noise [4,5].

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the fixed.qrMatrixSolve function by using fiaccel to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qrMatrixSolve -args {A,B,T.X,regularizationParameter} -o qrMatrixSolve_mex
```

Specify output type T.X and compute fixed-point $X = A\backslash B$ using the QR method.

```
X = qrMatrixSolve_mex(A,B,T.X,regularizationParameter);
```

Verify the Accuracy of the Output

Verify that the relative error between the fixed-point output and the output from MATLAB using the default double-precision floating-point values is small.

$$X_{\text{double}} = \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} \backslash \begin{bmatrix} 0_{n,\,p} \\ B \end{bmatrix}$$

```
A_lambda = double([regularizationParameter*eye(n);A]);
B_0 = [zeros(n,p);double(B)];
X_double = A_lambda\B_0;
relativeError = norm(X_double - double(X))/norm(X_double)

relativeError = 5.2634e-06
```

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Input Arguments

`m` — Number of rows in *A* and *B*
positive integer-valued scalar

Number of rows in A and B, specified as a positive integer-valued scalar.

Data Types: `double`

`n` — Number of columns in *A*
positive integer-valued scalar

Number of columns in A, specified as a positive integer-valued scalar.

Data Types: `double`

`max_abs_A` — Maximum of absolute value of *A*
scalar

Maximum of the absolute value of A, specified as a scalar.

Example: `max(abs(A(:)))`

Data Types: `double`

`max_abs_B` — Maximum of absolute value of *B*
scalar

Maximum of the absolute value of B, specified as a scalar.

Example: `max(abs(B(:)))`

Data Types: `double`

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision of the input and output, specified as a positive integer-valued scalar.

Data Types: `double`

`noiseStandardDeviation` — Standard deviation of additive random noise in *A*
scalar

Standard deviation of additive random noise in A, specified as a scalar.

If `noiseStandardDeviation` is not specified, then the default is the standard deviation of the complex-valued quantization noise $\sigma_q = \left(2^{-\text{precisionBits}}\right)/(\sqrt{6})$, which is calculated by `fixed.complexQuantizationNoiseStandardDeviation`.

Data Types: `double`

p_s — Probability that estimate of lower bound *s* is larger than the actual smallest singular value of the matrix

$\approx 3 \cdot 10^{-7}$ (default) | scalar

Probability that estimate of lower bound *s* is larger than the actual smallest singular value of the matrix, specified as a scalar. Use `fixed.complexSingularValueLowerBound` to estimate the smallest singular value, *s*, of *A*. If `p_s` is not specified, the default value is $p_s = (1/2) \cdot (1 + \mathrm{erf}(-5/\sqrt{2})) \approx 3 \cdot 10^{-7}$ which is 5 standard deviations below the mean, so the probability that the estimated bound for the smallest singular value is less than the actual smallest singular value is 1-p_s ≈ 0.9999997.

Data Types: `double`

`regularizationParameter` — Regularization parameter

`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

`regularizationParameter` is the Tikhonov regularization parameter of the least-squares problem $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`maxWordLength` — Maximum word length of fixed-point types

`128` (default) | positive integer

Maximum word length of fixed-point types, specified as a positive integer.

If the word length of the fixed-point type exceeds the specified maximum word length, the default of `128` bits is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

T — Fixed-point types for *A*, *B*, and *X*

struct

Fixed-point types for *A*, *B*, and *X*, returned as a struct. The struct `T` has fields `T.A`, `T.B`, and `T.X`. These fields contain `fi` objects that specify fixed-point types for

- *A* and *B* that guarantee no overflow will occur in the QR algorithm.

  The QR algorithm transforms *A* in-place into upper-triangular *R* and transforms *B* in-place into *C*=*Q*'*B*, where *QR*=*A* is the QR decomposition of *A*.
- *X* such that there is a low probability of overflow.

Tips

Use `fixed.complexQRMatrixSolveFixedpointTypes` to compute fixed-point types for the inputs of these functions and blocks.

- `fixed.qrMatrixSolve`
- Complex Burst Matrix Solve Using QR Decomposition
- Complex Partial-Systolic Matrix Solve Using QR Decomposition

Algorithms

`T.A` and `T.B` are computed using `fixed.qrFixedpointTypes`. The number of integer bits required to prevent overflow is derived from the following bounds on the growth of R and $C=Q'B$ [1]. The required number of integer bits is added to the number of bits of precision, `precisionBits`, of the input, plus one for the sign bit, plus one bit for intermediate CORDIC gain of approximately 1.6468 [2].

The elements of R are bounded in magnitude by

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

The elements of $C=Q'B$ are bounded in magnitude by

$$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|).$$

`T.X` is computed by bounding the output, X, in the least-squares solution of $AX=B$ using the following formula [3] [4].

The elements of $X=R\backslash(Q'B)$ are bounded in magnitude by

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}.$$

Computing the singular value decomposition to derive the above bound on X is more computationally expensive than the entire matrix solve, so the `fixed.complexSingularValueLowerBound` function is used to estimate a bound on `min(svd(A))`.

Version History

Introduced in R2021b

R2022b: Support for maximum word length

You can now use the `maxWordLenth` parameter to specify the maximum word length of the fixed-point types.

R2022a: Support for Tikhonov regularization parameter

The `fixed.complexQRMatrixSolveFixedpointTypes` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

References

[1] “Perform QR Factorization Using CORDIC”

[2] Voler, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers* EC-8 (1959): 330-334.

[3] Bryan, Thomas A. and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[4] Chen, Zizhong and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no. 3 (July 2005): 603-620.

See Also

Functions
`fixed.complexQuantizationNoiseStandardDeviation` | `fixed.complexSingularValueLowerBound` | `fixed.qrFixedpointTypes` | `fixed.qrMatrixSolve`

Blocks
Complex Burst Matrix Solve Using QR Decomposition | Complex Partial-Systolic Matrix Solve Using QR Decomposition

fixed.complexQuantizationNoiseStandardDeviation

Estimate standard deviation of quantization noise of complex-valued signal

Syntax

```
noiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(
precisionBits)
```

Description

`noiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBits)` returns an estimate of the quantization noise standard deviation of a complex-valued signal with a quantization level $q=2^{-precisionBits}$, where `precisionBits` is the required number of bits of precision.

Examples

Estimate Standard Deviation of Quantization Noise of Complex-Valued Signal

Quantizing a complex signal to p bits of precision can be modeled as a linear system that adds normally distributed noise with a standard deviation of $\zeta_{\text{noise}} = \frac{2^{-p}}{\sqrt{6}}$ [1,2].

Compute the theoretical quantization noise standard deviation with p bits of precision using the `fixed.complexQuantizationNoiseStandardDeviation` function.

```
p = 14;
theoreticalQuantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(p
```

The returned value is $\zeta_{\text{noise}} = \frac{2^{-p}}{\sqrt{6}}$.

Create a complex signal with n samples.

```
rng('default');
n = 1e6;
x = complex(rand(1,n),rand(1,n));
```

Quantize the signal with p bits of precision.

```
wordLength = 16;
x_quantized = quantizenumeric(x,1,wordLength,p);
```

Compute the quantization noise by taking the difference between the quantized signal and the original signal.

```
quantizationNoise = x_quantized - x;
```

Compute the measured quantization noise standard deviation.

```
measuredQuantizationNoiseStandardDeviation = std(quantizationNoise)
```

```
measuredQuantizationNoiseStandardDeviation = 2.4902e-05
```

Compare the actual quantization noise standard deviation to the theoretical and see that they are close for large values of *n*.

```
theoreticalQuantizationNoiseStandardDeviation
```

```
theoreticalQuantizationNoiseStandardDeviation = 2.4917e-05
```

References

1 Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

2 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

Input Arguments

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision, specified as a positive integer-valued scalar.

Data Types: `double`

Output Arguments

`noiseStandardDeviation` — Noise standard deviation
scalar

Noise standard deviation, returned as a scalar.

Tips

`fixed.complexQuantizationNoiseStandardDeviation` is used in these functions.

- `fixed.complexQRMatrixSolveFixedpointTypes`
- `fixed.complexQlessQRMatrixSolveFixedpointTypes`

Algorithms

The variance of a complex-valued error sequence $e(k)$ with quantization level q=$2^{\text{-}precisionBits}$ [1][2] is

$$\sigma_q^2 = \frac{2}{q}\int_{-q/2}^{q/2} e^2 de = \frac{q^2}{6} = \frac{2^{-2precisionBits}}{6}.$$

The standard deviation of a real error sequence $e(k)$ is

$$\sigma_q = \frac{2^{-precisionBits}}{\sqrt{6}}.$$

Version History

Introduced in R2021b

References

[1] Widrow, Bernard. "A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory." *IRE Transactions on Circuit Theory* 3, no. 4 (December 1956): 266-276.

[2] Widrow, Bernard, and Kollár, István. *Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge, UK: Cambridge University Press, 2008.

See Also

`fixed.complexQRMatrixSolveFixedpointTypes` | `fixed.complexQlessQRMatrixSolveFixedpointTypes`

fixed.complexSingularValueLowerBound

Estimate lower bound for smallest singular value of complex-valued matrix

Syntax

```
s_n = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n)
s_n = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n,
regularizationParameter)
```

Description

`s_n = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n)` returns an estimate of a lower bound, `s_n`, for the smallest singular value of a complex-valued matrix with `m` rows and `n` columns, where $m \geq n$.

`s_n = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n, regularizationParameter)` returns an estimate of a lower bound, `s_n`, for the smallest singular value of a complex-valued matrix $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the `regularizationParameter`, A is an m-by-n matrix with $m >= n$, and I_n = `eye(`n`)`.

`p_s_n` and `regularizationParameter` are optional parameters. If not supplied or empty, then their default values are used.

Examples

Algorithms to Determine Fixed-Point Types for Complex Q-less QR Matrix Solve A'AX=B

This example shows the algorithms that the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the complex matrix equation $A'AX = B$, where A is an m-by-n matrix with $m \geq n$, B is n-by-p, and X is n-by-p.

Overview

You can solve the fixed-point matrix equation $A'AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $R'RX = B$. To solve for X, compute $X = R\backslash(R'\backslash B)$ through forward- and backward-substitution of R into B.

You can determine appropriate fixed-point types for the matrix equation $A'AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.complexQlessQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on R, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of $R = Q'A$ is

$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|)$.

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \leq \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Since computing $\mathrm{svd}(A)$ is more computationally expensive than solving the system of equations, the `fixed.complexQlessQRMatrixSolveFixedpointTypes` function estimates a lower bound of $\min(\mathrm{svd}(A))$.

Fixed-point types for the solution of the matrix equation $(A'A)X = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a^2/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned}
||Av||_2 &\leq ||A||_2||v||_2 \\
||Q||_2 &= 1 \\
||v||_\infty &= \max(|v(:)|) \\
||v||_\infty &\leq ||v||_2 \leq \sqrt{m}||v||_\infty
\end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|)$.

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:, j) = Q'A(:, j)$, so

$$\begin{aligned}
\max(|R(:,j)|) &= ||R(:,j)||_\infty \\
&\le ||R(:,j)||_2 \\
&= ||Q'A(:,j)||_2 \\
&\le ||Q'||_2\,||A(:,j)||_2 \\
&= ||A(:,j)||_2 \\
&\le \sqrt{m}\,||A(:,j)||_\infty \\
&= \sqrt{m}\max(|A(:,j)|) \\
&\le \sqrt{m}\max(|A(:)|).
\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Upper Bound for X = (A'A)\B

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Proof of Upper Bound for X = (A'A)\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{n}\max(|B(:)|)/\min(\mathrm{svd}(A))^2 = \infty$ and so the inequality is true.

If $A'Ax = b$ and $QR = A$ is the economy-size QR decomposition of A, then $A'Ax = R'Q'QRx = R'Rx = b$. If A is full rank then $x = R^{-1} \cdot ((R')^{-1}b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the j th column of B. Then

$$\begin{aligned}
\max(|x(:)|) &= ||x||_\infty \\
&\le ||x||_2 \\
&= ||R^{-1} \cdot ((R')^{-1}b)||_2 \\
&\le ||R^{-1}||_2\,||(R')^{-1}||_2\,||b||_2 \\
&= \left(1/\min(\mathrm{svd}(A))^2\right) \cdot ||b||_2 \\
&= ||b||_2/\min(\mathrm{svd}(A))^2 \\
&\le \sqrt{n}\,||b||_\infty/\min(\mathrm{svd}(A))^2 \\
&= \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2.
\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of min(svd(A))for complex-valued A using the following formula,

$$s = \frac{\sigma_N}{\sqrt{2}}\sqrt{\gamma^{-1}\left(\frac{p_s\Gamma(m-n+2)^2\Gamma(n)}{\Gamma(m+1)\Gamma(m-n+1)(m-n+1)},\ m-n+1\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1-p_s$ is the probability that $s \le \min(\mathrm{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.4 from [3] and rearranging terms.

Since $s \le \min(\mathrm{svd}(A))$ with probability $1-p_s$, then you can bound the magnitude of the elements of X without computing svd(A),

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2} \le \frac{\sqrt{n}\max(|B(:)|)}{s^2} \text{ with probability } 1-p_s.$$

You can compute s using the `fixed.complexSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean, $p_s = (1+\mathrm{erf}(-5/\sqrt{2}))/2 \approx 2.8665 \cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1-p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, and $X = (A'A)\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices `B` and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, complex-valued matrices `A` and `B` are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing the real and imaginary parts of a complex signal is $2^{-\text{precisionBits}}/\sqrt{6}$ [4,5]. Use `fixed.complexQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBit
```

```
quantizationNoiseStandardDeviation = 2.4333e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.complexQlessQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.complexQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 27
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = (A'A)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 24
```

Upper Bound for R

The upper bound for R is computed using the formula $\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$, where m is the number of rows of matrix A. This upper bound is used to select a fixed-point type with the required number of bits of precision to avoid an overflow in the upper bound.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 24.4949
```

Lower Bound for min(svd(A)) for Complex A

A lower bound for min(svd(A)) is estimated by the `fixed.complexSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.complexSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviatio
```

```
estimatedSingularValueLowerBound = 0.0389
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,numSamples
    noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 24.4949
```

```
max(actualMaxR)
```

```
ans = 9.4990
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

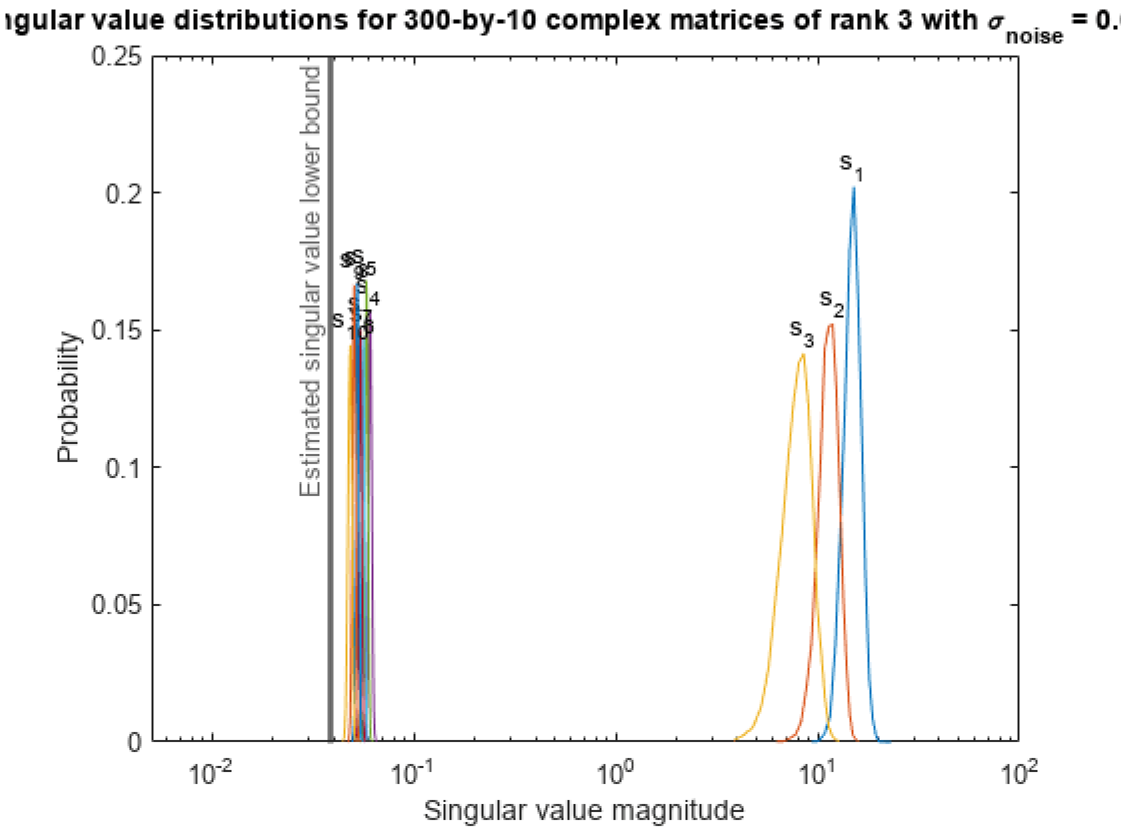
```
estimatedSingularValueLowerBound = 0.0389
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0443
```

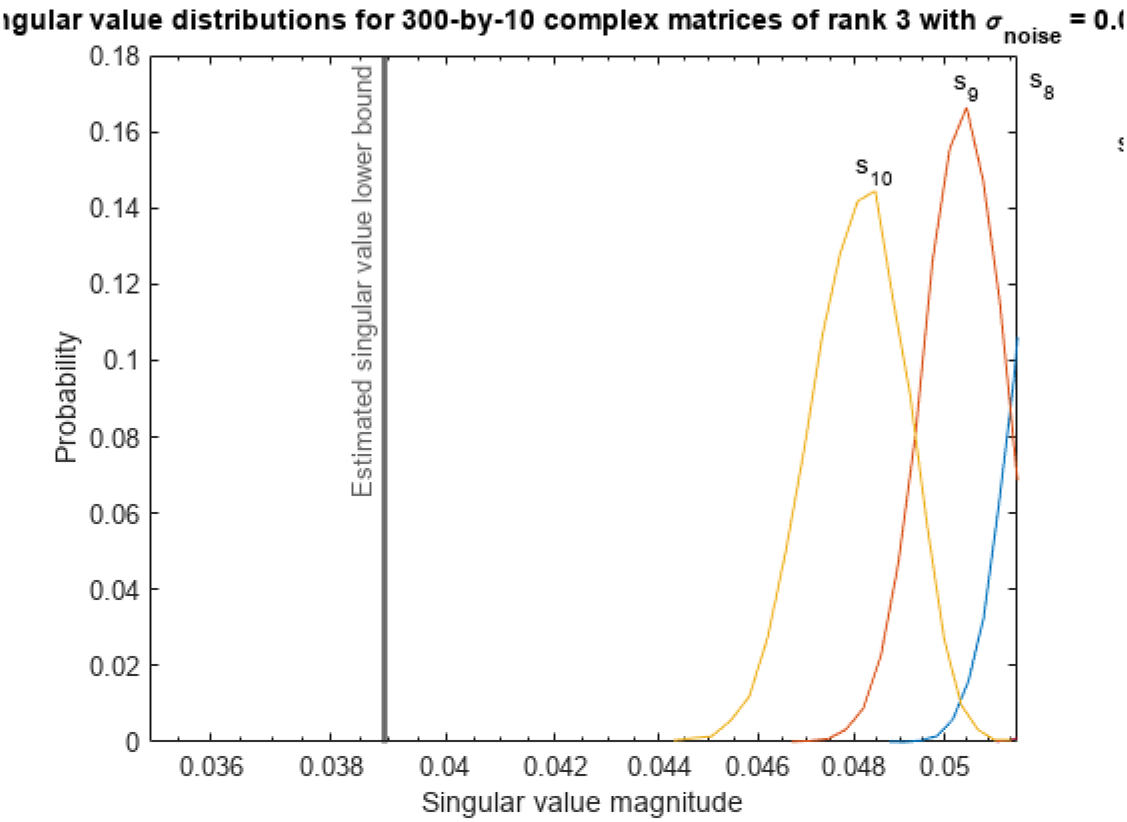
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,...
    noiseStandardDeviation,singularValues,...
    estimatedSingularValueLowerBound,"complex");
```



Zoom in to the smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.complexQlessQRMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDevia
```

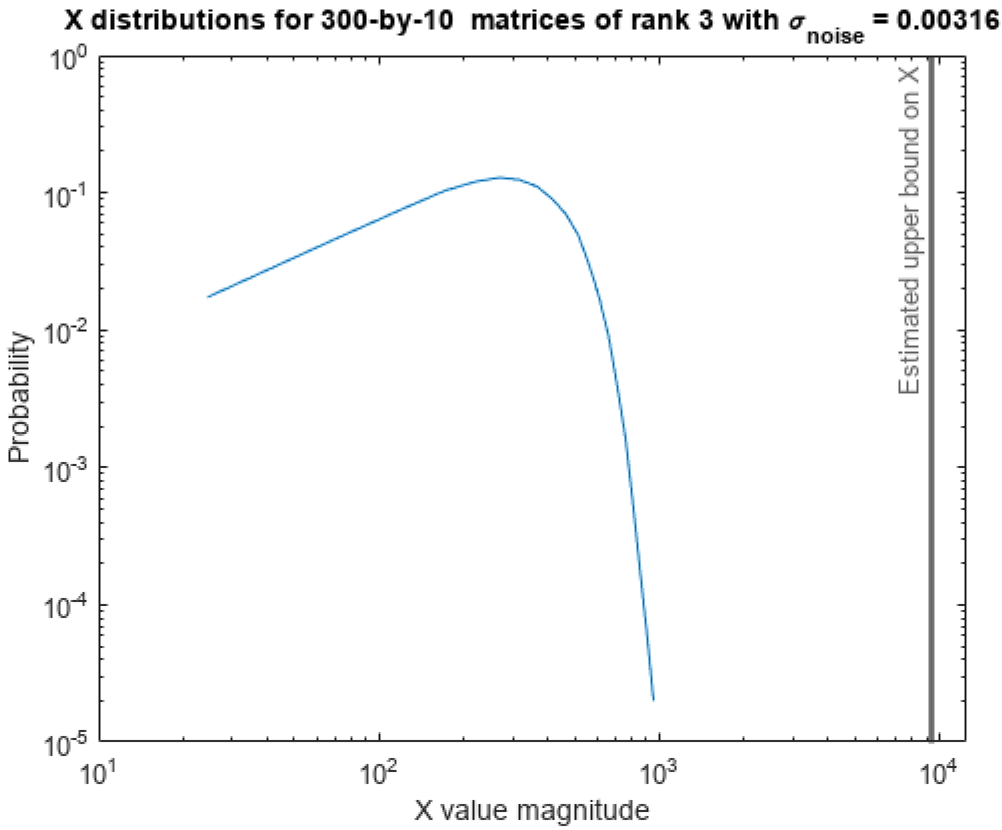
```
estimated_largest_X = 9.3348e+03
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 977.7440
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"complex normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $A'AX = B$. It returns the maximum values of $R = Q'A$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,...
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = (max_abs_A/sqrt(2))*fixed.example.complexRandomLowRankMatrix(m,n,rankA);
        % Adding random noise makes A non-singular.
        A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.complexUniformRandomArray(-max_abs_B,max_abs_B,n,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [~,R] = qr(A,0);
        X = R\(R'\B);
        actualMaxR(j) = max(abs(R(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
```

```
    end
end
```

References

1 Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.

2 “Perform QR Factorization Using CORDIC”. Derivation of the bound on growth when computing QR. MathWorks. 2010.

3 Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.

4 Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

5 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

6 Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Algorithms to Determine Fixed-Point Types for Complex Least-Squares Matrix Solve AX=B

This example shows the algorithms that the `fixed.complexQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the complex least-squares matrix equation $AX = B$, where A is an m-by-n matrix with $m \geq n$, B is m-by-p, and X is n-by-p.

Overview

You can solve the fixed-point least-squares matrix equation $AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, and transforms matrix B in-place to $C = Q'B$, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $RX = C$. To solve for X, compute $X = R\backslash C$ through back-substitution of R into C.

You can determine appropriate fixed-point types for the least-squares matrix equation $AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.complexQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on $R = Q'A$, $C = Q'B$, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of $R = Q'A$ is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

The upper bound for the magnitude of the elements of $C = Q'B$ is

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|).$$

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}.$$

Since computing svd(A) is more computationally expensive than solving the system of equations, the `fixed.complexQRMatrixSolveFixedpointTypes` function estimates a lower bound of min(svd(A)).

Fixed-point types for the solution of the matrix equation $AX = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned} ||Av||_2 &\le ||A||_2||v||_2 \\ ||Q||_2 &= 1 \\ ||v||_\infty &= \max(|v(:)|) \\ ||v||_\infty &\le ||v||_2 \le \sqrt{m}||v||_\infty \end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:, j) = Q'A(:, j)$, so

$$\begin{aligned}\max(|R(:,j)|) &= ||R(:,j)||_\infty \\ &\le ||R(:,j)||_2 \\ &= ||Q'A(:,j)||_2 \\ &\le ||Q'||_2\,||A(:,j)||_2 \\ &= ||A(:,j)||_2 \\ &\le \sqrt{m}\,||A(:,j)||_\infty \\ &= \sqrt{m}\max(|A(:,j)|) \\ &\le \sqrt{m}\max(|A(:)|).\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Upper Bound for C = Q'B

The upper bound for the magnitude of the elements of $C = Q'B$ is

$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|)$.

Proof of Upper Bound for C = Q'B

The proof of the upper bound for $C = Q'B$ is the same as the proof of the upper bound for $R = Q'A$ by substituting C for R and B for A.

Upper Bound for X = A\B

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$\max(|X(:)|) \le \dfrac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}$.

Proof of Upper Bound for X = A\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{m}\max(|B(:)|)/\min(\mathrm{svd}(A)) = \infty$ and so the inequality is true.

If A is full rank, then $x = R^{-1}(Q'b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the jth column of B. Then

$$\begin{aligned}\max(|x(:)|) &= ||x||_\infty \\ &\le ||x||_2 \\ &= ||R^{-1}\cdot(Q'b)||_2 \\ &\le ||R^{-1}||_2\,||Q'||_2\,||b||_2 \\ &= (1/\min(\mathrm{svd}(A)))\cdot 1\cdot ||b||_2 \\ &= ||b||_2/\min(\mathrm{svd}(A)) \\ &\le \sqrt{m}\,||b||_\infty/\min(\mathrm{svd}(A)) \\ &= \sqrt{m}\max(|b(:)|)/\min(\mathrm{svd}(A)).\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{m}\max(|b(:)|)/\min(\text{svd}(A))$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\text{svd}(A))$for complex-valued A using the following formula,

$$s = \frac{\sigma_N}{\sqrt{2}}\sqrt{\gamma^{-1}\left(\frac{p_s\Gamma(m-n+2)^2\Gamma(n)}{\Gamma(m+1)\Gamma(m-n+1)(m-n+1)},\ m-n+1\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1 - p_s$ is the probability that $s \le \min(\text{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.4 from [3] and rearranging terms.

Since $s \le \min(\text{svd}(A))$ with probability $1 - p_s$, then you can bound the magnitude of the elements of X without computing $\text{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))} \le \frac{\sqrt{m}\max(|B(:)|)}{s} \text{ with probability } 1 - p_s.$$

You can compute s using the `fixed.complexSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean,
$p_s = (1 + \text{erf}(-5/\sqrt{2}))/2 \approx 2.8665 \cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1 - p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, $C = Q'B$, and $X = A\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrix `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, complex-valued matrices A and B are constructed such that the magnitude of the real and imaginary parts of their elements is less than or equal to one, so the maximum possible absolute value of any element is $|1 + 1i| = \sqrt{2}$. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of A and B.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = sqrt(2);
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = sqrt(2);
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing the real and imaginary parts of a complex signal is $2^{-\text{precisionBits}}/\sqrt{6}$ [4,5]. Use the `fixed.complexQuantizationNoiseStandardDeviation` function to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.complexQuantizationNoiseStandardDeviation(precisionBi
```

```
quantizationNoiseStandardDeviation = 2.4333e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.complexQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.complexQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.B` is the type computed for transforming B to $Q'B$ in-place so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = A\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 37
        FractionLength: 24
```

Upper Bounds for R and C=Q'B

The upper bounds for R and $C = Q'B$ are computed using the following formulas, where m is the number of rows of matrices A and B.

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|)$$

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|)$$

These upper bounds are used to select a fixed-point type with the required number of bits of precision to avoid overflows.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 24.4949
```

```
upperBoundQB = sqrt(m)*max_abs_B
```

```
upperBoundQB = 24.4949
```

Lower Bound for min(svd(A)) for Complex A

A lower bound for min(svd(A)) is estimated by the `fixed.complexSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.complexSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.complexSingularValueLowerBound(m,n,noiseStandardDeviatio
```

```
estimatedSingularValueLowerBound = 0.0389
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B
    numSamples,noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 24.4949
```

```
max(actualMaxR)
```

```
ans = 9.6720
```

You can see that the upper bound on $C = Q'B$ compared to the measured simulation results of the maximum value of $C = Q'B$ over all runs is also within an order of magnitude.

```
upperBoundQB
```

```
upperBoundQB = 24.4949
```

```
max(actualMaxQB)
```

```
ans = 4.4764
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

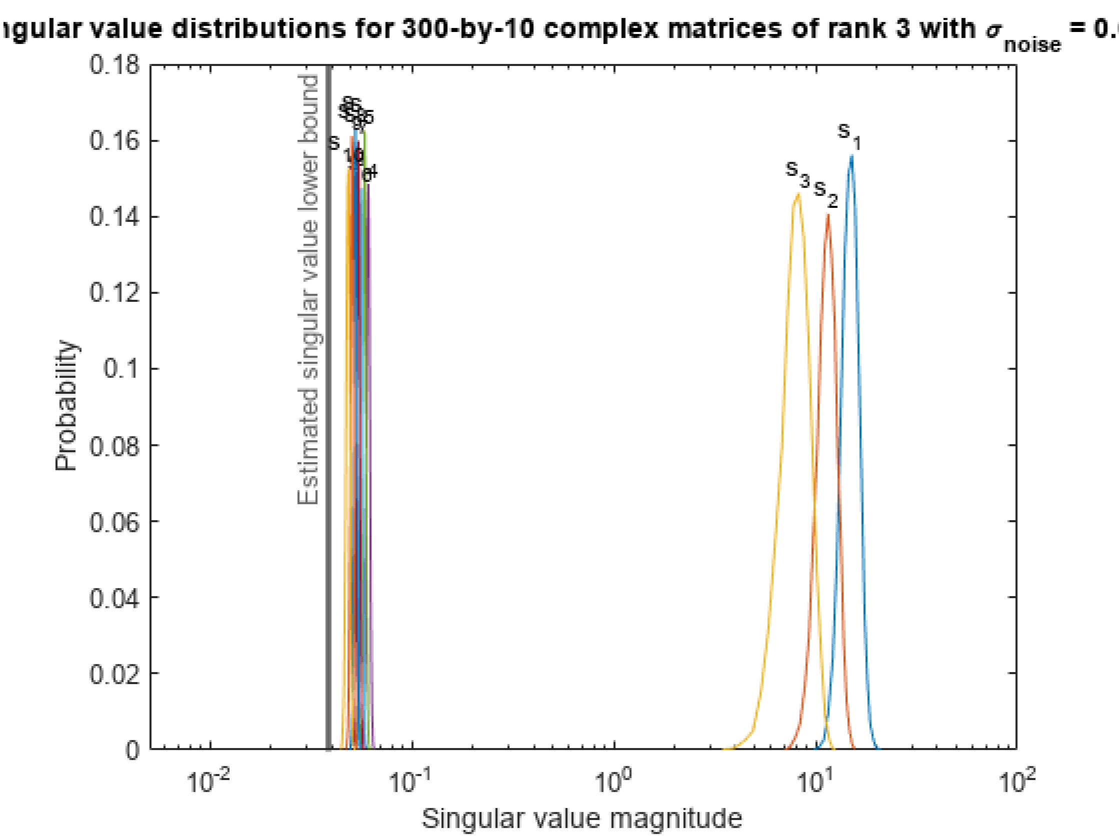
```
estimatedSingularValueLowerBound = 0.0389
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0443
```

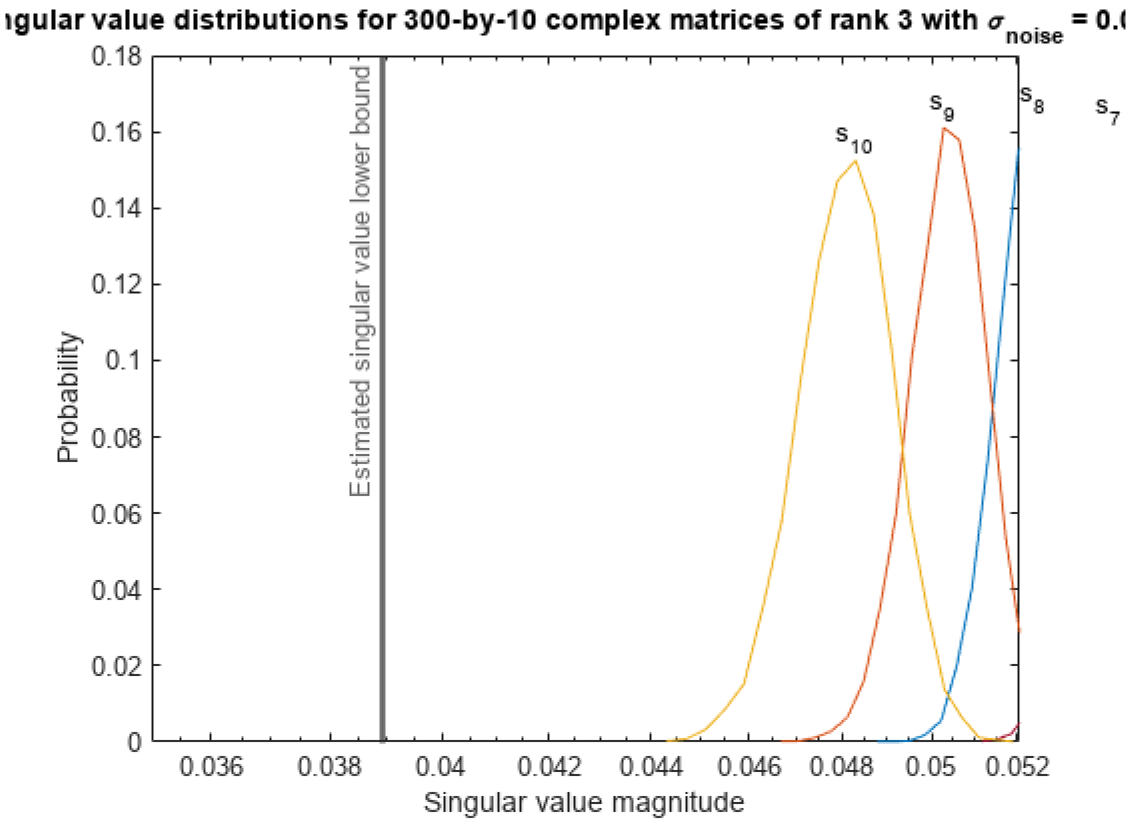
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,noiseStandardDeviation,...
    singularValues,estimatedSingularValueLowerBound,"complex");
```



Zoom in to the smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.complexMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDeviation)
```

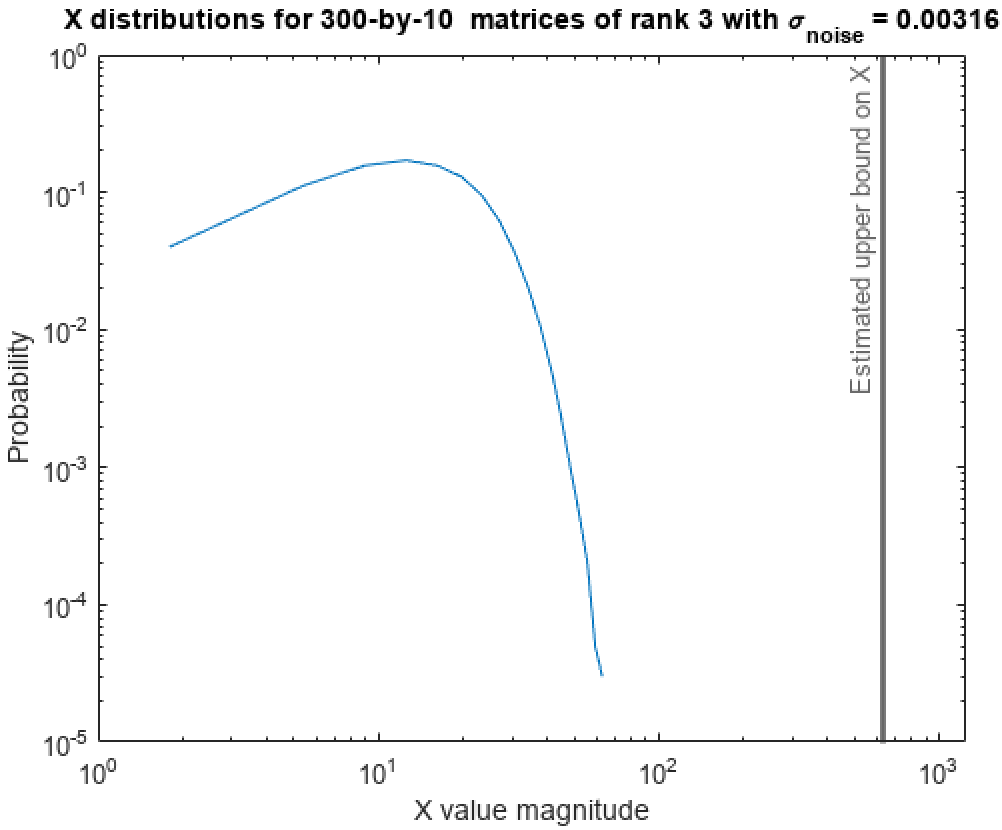
```
estimated_largest_X = 629.3194
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 70.2644
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"complex normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $AX = B$. It returns the maximum values of $R = Q'A$ and $C = Q'B$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    actualMaxQB = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = (max_abs_A/sqrt(2))*fixed.example.complexRandomLowRankMatrix(m,n,rankA);
        % Adding normally distributed random noise makes A non-singular.
        A = A + fixed.example.complexNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.complexUniformRandomArray(-max_abs_B,max_abs_B,m,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [Q,R] = qr(A,0);
        C = Q'*B;
        X = R\C;
        actualMaxR(j) = max(abs(R(:)));
```

```
        actualMaxQB(j) = max(abs(C(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
    end
end
```

References

1 Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.

2 Perform QR Factorization Using CORDIC. Derivation of the bound on growth when computing QR. MathWorks. 2010.

3 Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.

4 Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

5 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

6 Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Input Arguments

m — Number of rows in matrix
positive integer-valued scalar

Number of rows in matrix, specified as a positive integer-valued scalar. The number of rows, `m`, must be greater than or equal to the number of columns, `n`.

Data Types: `double`

n — Number of columns in matrix
positive integer-valued scalar

Number of columns in matrix, specified as a positive integer-valued scalar. The number of rows, `m`, must be greater than or equal to the number of columns, `n`.

Data Types: `double`

noiseStandardDeviation — Standard deviation of additive random noise in matrix
scalar

Standard deviation of additive random noise in matrix, specified as a scalar.

Data Types: `double`

p_s_n — Probability that estimate of lower bound is larger than actual smallest singular value of matrix
2.8665e-07 (default) | scalar

Probability that estimate of lower bound is larger than actual smallest singular value of matrix, specified as a scalar.

If `p_s_n` is not supplied or empty, then the default of `p_s = (1/2)*(1+erf(-5/sqrt(2))) = 2.8665e-07` is used, which is 5 standard deviations below the mean, so the probability that the estimated lower bound for the smallest singular value is less than the actual smallest singular value is `1 - p_s = 0.99999971 - p_s = 0.9999997`.

Data Types: `double`

regularizationParameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

`regularizationParameter` is the Tikhonov regularization parameter of the matrix $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ where λ is the `regularizationParameter`, A is an m-by-n matrix with $m >= n$, and I = `eye(`n`)`..

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

s_n — Estimate of lower bound for smallest singular value of complex-valued matrix
scalar

Estimate of lower bound for smallest singular value of complex-valued matrix, returned as a scalar.

Tips

- Use `fixed.complexSingularValueLowerBound` to used estimate the smallest singular value of a matrix to estimate a bound for $\max(|X(:)|)$. For example, in `fixed.complexQRMatrixSolveFixedpointTypes`, the elements of $X=R\backslash(Q'B)$ are bounded in magnitude by

  $$\max(|X(:)|) \leq \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))} \leq \frac{\sqrt{m}\max(|B(:)|)}{s}$$

  with probability 1-p_s.
- $\max(|X(:)|)$ is smaller when the denominator in the above equation is larger.
- If nothing else is known about a matrix, then in general the smallest singular value will be larger if:

  - there is additive random noise.

  - the number of rows, m, is much larger than the number of columns, n.
- If the noise standard deviation is not known, you can approximate it as the standard deviation of the quantization error. You can compute the quantization error using `fixed.complexQuantizationNoiseStandardDeviation`.
- For s to be a useful bound on the smallest singular value of A, the probability that s is greater than the smallest singular value of A should be small. A practical value to use is

  $$p_s = (1/2) \cdot (1 + \mathrm{erf}(-5/\sqrt{2})) \approx 3 \cdot 10^{-7}$$

  which is 5 standard deviations below the mean, so the probability that the estimated bound for the smallest singular value is less than the actual smallest singular value is 1-p_s ≈ 0.9999997.
- `fixed.complexSingularValueLowerBound` is used in these functions.
  - `fixed.complexQRMatrixSolveFixedpointTypes`
  - `fixed.complexQlessQRMatrixSolveFixedpointTypes`

Algorithms

Given a m-by-n complex-valued matrix A and standard deviation σ_N of additive random noise on the elements of A, you can compute an estimate of a lower bound for the smallest singular value of A, s, such that the probability, p_s, of s being greater than the smallest singular value of A using this formula [1][2].

$$s = \frac{\sigma_N}{\sqrt{2}}\sqrt{\gamma^{-1}\left(\frac{p_s\Gamma(m-n+2)^2\Gamma(n)}{\Gamma(m+1)\Gamma(m-n+1)(m-n+1)}, m-n+1\right)}$$

Version History

Introduced in R2021b

R2022a: Support for Tikhonov regularization parameter

The `fixed.complexSingularValueLowerBound` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

References

[1] Bryan, Thomas A. and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[2] Chen, Zizhong and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no. 3 (July 2005): 603-620. https://doi.org/10.1137/040616413.

See Also

`fixed.complexQRMatrixSolveFixedpointTypes` | `fixed.complexQuantizationNoiseStandardDeviation` |

`fixed.complexQRMatrixSolveFixedpointTypes` | `fixed.complexQlessQRMatrixSolveFixedpointTypes`

fixed.cordicDivide

Fixed-point divide using CORDIC

Syntax

```
y = fixed.cordicDivide(num,den,OutputType)
```

Description

`y = fixed.cordicDivide(num,den,OutputType)` divides `num` by `den` using the output data type specified by `OutputType`.

Examples

Divide Using CORDIC

```
num = fi(1);
den = fi(10);
OutputType = fi([],1,16,15);
y = fixed.cordicDivide(num,den,OutputType)
```

```
y = 

    0.1000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Input Arguments

`num` — Numerator
scalar | vector | matrix | multidimensional array

Numerator, specified as a real-valued scalar, vector, matrix, or multidimensional array.

- If `num` is a floating-point type, `den` must also be a floating-point type and `OutputType` must specify a floating-point data type.
- If `num` is a built-in integer type, `den` must also be a built-in integer type and `OutputType` must specify a built-in integer data type.
- If `num` is a fixed-point type, `den` must also be a fixed-point type and `OutputType` must specify a fixed-point data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

den — Denominator
scalar | vector | matrix | multidimensional array

Numerator, specified as a real-valued scalar, vector, matrix, or multidimensional array.

- If `num` is a floating-point type, `den` must also be a floating-point type and `OutputType` must specify a floating-point data type.
- If `num` is a built-in integer type, `den` must also be a built-in integer type and `OutputType` must specify a built-in integer data type.
- If `num` is a fixed-point type, `den` must also be a fixed-point type and `OutputType` must specify a fixed-point data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`OutputType` — Data type of output
`fi` object | `numerictype` object | `Simulink.NumericType` object

Data type of the output, specified as a `fi` object, `numerictype`, or `Simulink.NumericType` object.

- If `num` is a floating-point type, `den` must also be a floating-point type and `OutputType` must specify a floating-point data type.
- If `num` is a built-in integer type, `den` must also be a built-in integer type and `OutputType` must specify a built-in integer data type.
- If `num` is a fixed-point type, `den` must also be a fixed-point type and `OutputType` must specify a fixed-point data type.

Example: `fi([],1,16,15)`

Example: `numerictype(1,16,15)`

Example: `fixdt(1,16,15)`

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Algorithms

For fixed-point inputs `num` and `den`, `fixed.cordicDivide` wraps on overflow for division by zero. The behavior for fixed-point division by zero is summarized in the table below.

Wrap Overflow	Saturate Overflow
0/0 = 0	0/0 = 0
1/0 = 0	1/0 = upper bound
-1/0 = 0	-1/0 = lower bound

For floating-point inputs, `fixed.cordicDivide` follows IEEE Standard 754.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.cordicReciprocal` | Real Divide HDL Optimized | Complex Divide HDL Optimized | Real Reciprocal HDL Optimized

fixed.cordicReciprocal

Fixed-point reciprocal using CORDIC

Syntax

`y = fixed.cordicReciprocal(u,OutputType)`

Description

`y = fixed.cordicReciprocal(u,OutputType)` returns `1./u` with the output cast to the data type specified by `OutputType`.

Examples

Reciprocal Using CORDIC

```
u = fi(10);
outputType = fi([],1,32,24);
y = fixed.cordicReciprocal(u,outputType)
```

```
y = 

    0.1000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

Input Arguments

u — Value to take reciprocal of
scalar | vector | matrix | multidimensional array

Value to take reciprocal of, specified as a scalar, vector, matrix, or multidimensional array.

- If `u` is a floating-point type, then `OutputType` must specify a floating-point data type.
- If `u` is a built-in integer type, then `OutputType` must specify a built-in integer data type.
- If `u` is a fixed-point type, then `OutputType` must specify a fixed-point data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`OutputType` — Data type of output
`fi` object | `numerictype` object | `Simulink.NumericType` object

Data type of the output, specified as a `fi` object, `numerictype`, or `Simulink.NumericType` object.

- If `num` is a floating-point type, `den` must also be a floating-point type and `OutputType` must specify a floating-point data type.
- If `num` is a built-in integer type, `den` must also be a built-in integer type and `OutputType` must specify a built-in integer data type.
- If `num` is a fixed-point type, `den` must also be a fixed-point type and `OutputType` must specify a fixed-point data type.

Example: `fi([],1,16,15)`

Example: `numerictype(1,16,15)`

Example: `fixdt(1,16,15)`

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so increases the expense of the computation and adds latency.

Algorithms

For fixed-point input `u`, `fixed.cordicReciprocal` wraps on overflow for division by zero. The behavior for fixed-point division by zero is summarized in the table below.

Wrap Overflow	Saturate Overflow
0/0 = 0	0/0 = 0
1/0 = 0	1/0 = upper bound
-1/0 = 0	-1/0 = lower bound

For floating-point inputs, `fixed.cordicReciprocal` follows IEEE Standard 754.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.cordicDivide` | Real Reciprocal HDL Optimized | Real Divide HDL Optimized | Complex Divide HDL Optimized

fixed.fimathLike

Return `fimath` object like the input

Syntax

```
F = fixed.fimathLike(X)
F = fixed.fimathLike(X,roundingMethod)
F = fixed.fimathLike(X,roundingMethod,overflowAction)
```

Description

`F = fixed.fimathLike(X)` returns `fimath` object `F` with `ProductMode` and `SumMode` set to the same fixed-point arithmetic properties as the input `X`, where `X` is a `fi` object or `numerictype` object.

By default, `RoundingMethod` is set to `'Floor'` and `OverflowAction` is set to `'Wrap'`. For fixed-point inputs, this function supports binary-point scaling and slope-bias scaling. If `X` is not a `fi` object or `numerictype` object, then empty is returned.

`F = fixed.fimathLike(X,roundingMethod)` specifies the rounding method to use.

`F = fixed.fimathLike(X,roundingMethod,overflowAction)` specifies the rounding method and overflow action to use.

Examples

Set `fimath` to be Like Input

Use `fixed.fimathLike` and `setfimath` to set the `fimath` fixed-point math settings to be like the input.

Define a fixed-point `fi` object.

```
X = fi(1,1,8,0)
```

Create a `fimath` object with the same fixed-point math settings as the input.

```
F = fixed.fimathLike(X)
```

```
F = 


        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: SpecifyPrecision
     ProductWordLength: 8
 ProductFractionLength: 0
               SumMode: SpecifyPrecision
         SumWordLength: 8
     SumFractionLength: 0
         CastBeforeSum: true
```

Apply the fixed-point math settings.

```
X(:) = setfimath(X,F) + 1

X = 

     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

If X is a `fi` object or `numerictype` object, then `fixed.fimathLike` returns a `fimath` object. Otherwise, `fixed.fimathLike` returns empty.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

`roundingMethod` — Rounding method to use
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method to use, specified as one of these values:

- `'Ceiling'` — Round toward positive infinity.
- `'Convergent'` — Round toward nearest. Ties round to the nearest even stored integer (least biased).
- `'Floor'` — Round toward negative infinity.
- `'Nearest'` — Round toward nearest. Ties round toward positive infinity.
- `'Round'` — Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
- `'Zero'` — Round toward zero.

Data Types: `char` | `string`

`overflowAction` — Action to take on overflow
`'Wrap'` (default) | `'Saturate'`

Action to take on overflow, specified as one of these values:

- `'Wrap'` — Wrap on overflow. This mode is also known as two's complement overflow.
- `'Saturate'` — Saturate to the maximum or minimum value of the fixed-point range on overflow.

Data Types: `char` | `string`

Output Arguments

F — Fixed-point math settings
`fimath` object

Fixed-point math settings, returned as a `fimath` object.

If X is not a `fi` object or `numerictype` object, then `fixed.fimathLike` returns empty.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`setfimath` | `fimath` | `fi` | `numerictype`

fixed.forgettingFactor

Compute forgetting factor required for streaming input data

Syntax

```
alpha = fixed.forgettingFactor(m)
```

Description

`alpha = fixed.forgettingFactor(m)` returns the forgetting factor α for an infinite number of rows with the equivalent gain of a matrix A with m rows.

Examples

Compute Forgetting Factor Required for Streaming Input Data

This example shows how to use the `fixed.forgettingFactor` and `fixed.forgettingFactorInverse` functions.

The growth in the QR decomposition can be seen by looking at the magnitude of the first element $R(1,1)$of the upper-triangular factor R, which is equal to the Euclidean norm of the first column of matrix A,

$|R(1,1)| = ||A(:,1)||_2$.

To see this, create matrix A as a column of ones of length n and compute R of the economy-size QR decomposition.

```
n = 1e4;
A = ones(n,1);
```

Then $|R(1,1)| = ||A(:,1)||_2 = \sqrt{\sum_{i=1}^{n} 1^2} = \sqrt{n}$.

```
R = fixed.qlessQR(A)
```

```
R = 100.0000
```

```
norm(A)
```

```
ans = 100
```

```
sqrt(n)
```

```
ans = 100
```

The diagonal elements of the upper-triangular factor R of the QR decomposition may be positive, negative, or zero, but `fixed.qlessQR` and `fixed.qrAB` always return the diagonal elements of R as non-negative.

In a real-time application, such as when data is streaming continuously from a radar array, you can update the QR decomposition with an exponential forgetting factor α where $0 < \alpha < 1$. Use the `fixed.forgettingFactor` function to compute a forgetting factor α that acts as if the matrix were being integrated over m rows to maintain a gain of about $\sqrt{m}$. The relationship between α and m is $\alpha = e^{-1/(2m)}$.

```
m = 16;
alpha = fixed.forgettingFactor(m);
R_alpha = fixed.qlessQR(A,alpha)
```

```
R_alpha = 3.9377
```

```
sqrt(m)
```

```
ans = 4
```

If you are working with a system and have been given a forgetting factor α, and want to know the effective number of rows m that you are integrating over, then you can use the `fixed.forgettingFactorInverse` function. The relationship between m and α is $m = \frac{-1}{2\log(\alpha)}$.

```
fixed.forgettingFactorInverse(alpha)
```

```
ans = 16
```

Input Arguments

`m` — Number of rows in matrix *A*
positive integer-valued scalar

Number of rows in matrix A, specified as a positive integer-valued scalar.

Data Types: `double`

Output Arguments

`alpha` — Forgetting factor
scalar

Forgetting factor, returned as a scalar.

Tips

Use `fixed.forgettingFactor` to compute a forgetting factor for these functions and blocks.

- `fixed.qlessQR`
- `fixed.qlessQRMatrixSolve`
- Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor
- Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor
- Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Algorithms

In real-time applications, such as when data is streaming continuously from a radar array [1], the QR decomposition is often computed continuously as each new row of data arrives. In these systems, the previously computed upper-triangular matrix, R, is updated and weighted by forgetting factor α, where $0 < \alpha < 1$. This computation treats the matrix A as if it is infinitely tall. The series of transformations is as follows.

$$R_0 = \text{zeros}(n, n)$$

$$\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix} \rightarrow \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \alpha R_1 \\ A(2,:) \end{bmatrix} \rightarrow \begin{bmatrix} R_2 \\ 0 \end{bmatrix}$$

$$\vdots$$

$$\begin{bmatrix} \alpha R_k \\ A(k,:) \end{bmatrix} \rightarrow \begin{bmatrix} R_{k+1} \\ 0 \end{bmatrix}$$

Without the forgetting factor α, the values of R would grow without bound.

With the forgetting factor, the gain in R is

$$g = \sqrt{\frac{1}{2}\int_0^\infty \alpha^x dx} = \sqrt{\frac{-1}{2\log(\alpha)}}.$$

The gain of computing R without a forgetting factor from an m-by-n matrix A is $\sqrt{m}$. Therefore,

$$\sqrt{m} = \sqrt{\frac{-1}{2\log(\alpha)}}$$

$$m = \frac{-1}{2\log(\alpha)}$$

$$\alpha = e^{-1/(2m)}.$$

Version History

Introduced in R2021b

References

[1] Rader, C.M. "VLSI Systolic Arrays for Adaptive Nulling." *IEEE Signal Processing Magazine* (July 1996): 29-49.

See Also

Functions

`fixed.qlessQR` | `fixed.qlessQRMatrixSolve` | `fixed.forgettingFactorInverse`

Blocks

Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic Matrix Solve Using Q-less QR

Decomposition with Forgetting Factor | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

fixed.forgettingFactorInverse

Compute the inverse of the forgetting factor required for streaming input data

Syntax

```
m = fixed.forgettingFactorInverse(alpha)
```

Description

`m = fixed.forgettingFactorInverse(alpha)` returns the number of rows with the equivalent gain of a matrix A with m rows, given a forgetting factor α.

Examples

Compute Forgetting Factor Required for Streaming Input Data

This example shows how to use the `fixed.forgettingFactor` and `fixed.forgettingFactorInverse` functions.

The growth in the QR decomposition can be seen by looking at the magnitude of the first element $R(1,1)$of the upper-triangular factor R, which is equal to the Euclidean norm of the first column of matrix A,

$|R(1,1)| = ||A(:,1)||_2$.

To see this, create matrix A as a column of ones of length n and compute R of the economy-size QR decomposition.

```
n = 1e4;
A = ones(n,1);
```

Then $|R(1,1)| = ||A(:,1)||_2 = \sqrt{\sum_{i=1}^{n} 1^2} = \sqrt{n}$.

```
R = fixed.qlessQR(A)
```

```
R = 100.0000
```

```
norm(A)
```

```
ans = 100
```

```
sqrt(n)
```

```
ans = 100
```

The diagonal elements of the upper-triangular factor R of the QR decomposition may be positive, negative, or zero, but `fixed.qlessQR` and `fixed.qrAB` always return the diagonal elements of R as non-negative.

In a real-time application, such as when data is streaming continuously from a radar array, you can update the QR decomposition with an exponential forgetting factor α where $0 < \alpha < 1$. Use the `fixed.forgettingFactor` function to compute a forgetting factor α that acts as if the matrix were being integrated over m rows to maintain a gain of about $\sqrt{m}$. The relationship between α and m is $\alpha = e^{-1/(2m)}$.

```
m = 16;
alpha = fixed.forgettingFactor(m);
R_alpha = fixed.qlessQR(A,alpha)
```

```
R_alpha = 3.9377
```

```
sqrt(m)
```

```
ans = 4
```

If you are working with a system and have been given a forgetting factor α, and want to know the effective number of rows m that you are integrating over, then you can use the `fixed.forgettingFactorInverse` function. The relationship between m and α is $m = \frac{-1}{2\log(\alpha)}$.

```
fixed.forgettingFactorInverse(alpha)
```

```
ans = 16
```

Input Arguments

`alpha` — Forgetting factor
scalar

Forgetting factor, specified as a scalar.

Data Types: `double`

Output Arguments

`m` — Number of rows in matrix *A*
positive integer-valued scalar

Number of rows in matrix A with the equivalent gain, returned as a positive integer-valued scalar.

Algorithms

In real-time applications, such as when data is streaming continuously from a radar array [1], the QR decomposition is often computed continuously as each new row of data arrives. In these systems, the previously computed upper-triangular matrix, R, is updated and weighted by forgetting factor α, where $0 < \alpha < 1$. This computation treats the matrix A as if it is infinitely tall. The series of transformations is as follows.

$$
\begin{aligned}
&R_0 = \text{zeros}(n, n)\\
&\begin{bmatrix} R_0 \\ A(1,:) \end{bmatrix} \rightarrow \begin{bmatrix} R_1 \\ 0 \end{bmatrix}\\
&\begin{bmatrix} \alpha R_1 \\ A(2,:) \end{bmatrix} \rightarrow \begin{bmatrix} R_2 \\ 0 \end{bmatrix}\\
&\vdots\\
&\begin{bmatrix} \alpha R_k \\ A(k,:) \end{bmatrix} \rightarrow \begin{bmatrix} R_{k+1} \\ 0 \end{bmatrix}
\end{aligned}
$$

Without the forgetting factor α, the values of R would grow without bound.

With the forgetting factor, the gain in R is

$$
g = \sqrt{\frac{1}{2}\int_0^\infty \alpha^x dx} = \sqrt{\frac{-1}{2\log(\alpha)}}.
$$

The gain of computing R without a forgetting factor from an m-by-n matrix A is $\sqrt{m}$. Therefore,

$$
\begin{aligned}
&\sqrt{m} = \sqrt{\frac{-1}{2\log(\alpha)}}\\
&m = \frac{-1}{2\log(\alpha)}\\
&\alpha = e^{-1/(2m)}.
\end{aligned}
$$

Version History

Introduced in R2021b

References

[1] Rader, C.M. "VLSI Systolic Arrays for Adaptive Nulling." *IEEE Signal Processing Magazine* (July 1996): 29-49.

See Also

Functions

`fixed.qlessQR` | `fixed.qlessQRMatrixSolve` | `fixed.forgettingFactor`

Blocks

Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor | Complex Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

fixed.forwardSubstitute

Solve lower-triangular system of equations through forward substitution

Syntax

```
x = fixed.forwardSubstitute(R, B)
x = fixed.forwardSubstitute(R, B, outputType)
```

Description

`x = fixed.forwardSubstitute(R, B)` performs forward substitution on upper-triangular matrix `R` to compute $x = R'\backslash B$.

`x = fixed.forwardSubstitute(R, B, outputType)` returns $x = R'\backslash B$, where the data type of output variable, `x`, is specified by `outputType`.

Examples

Solve a System of Equations Using Forward and Backward Substitution

This example shows how to solve the system of equations $(A'A)x = B$ using forward and backward substitution.

Specify the input variables, `A` and `B`.

```
rng default;
A = gallery('randsvd', [5,3], 1000);
b = [1; 1; 1; 1; 1];
```

Compute the upper-triangular factor, `R`, of `A`, where $A = QR$.

```
R = fixed.qlessQR(A);
```

Use forward and backward substitution to compute the value of `X`.

```
X = fixed.forwardSubstitute(R,b);
X(:) = fixed.backwardSubstitute(R,X)
```

```
X = 5×1
10^5 ×

   -0.9088
    2.7123
   -0.8958
         0
         0
```

This solution is equivalent to using the `fixed.qlessQRMatrixSolve` function.

```
x = fixed.qlessQRMatrixSolve(A,b)
```

```
x = 5×1
10⁵ ×

   -0.9088
    2.7123
   -0.8958
         0
         0
```

Input Arguments

R — Upper-triangular input matrix
matrix

Upper triangular input, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

B — Linear system factor
matrix

Linear system factor, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`outputType` — Output data type
`numerictype` object | numeric variable

Output data type, specified as a `numerictype` object or a numeric variable. If `outputType` is specified as a `numerictype` object, the output, `x`, will have the specified data type. If `outputType` is specified as a numeric variable, `x` will have the same data type as the numeric variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi` | `numerictype`

Output Arguments

x — Solution
matrix

Solution, returned as a matrix satisfying the equation $x = R'\backslash B$.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

R and *B* must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also
`fixed.backwardSubstitute` | `fixed.qlessQR` | `fixed.qlessQRUpdate` | `fixed.qrAB` | `fixed.qrMatrixSolve` | `fixed.qlessQRMatrixSolve`

fixed.jacobiSVD

Fixed-point Jacobi singular value decomposition

Syntax

```
S = fixed.jacobiSVD(A)
[U,s,V] = fixed.jacobiSVD(A)
[ ___ ] = fixed.jacobiSVD( ___ ,numberOfSweeps)
```

Description

`S = fixed.jacobiSVD(A)` returns a vector containing the singular values of matrix `A` in descending order.

`[U,s,V] = fixed.jacobiSVD(A)` performs a singular value decomposition of matrix `A` such that `A = U*diag(s)*V'`. `s` is a vector of nonnegative elements in decreasing order. `U` and `V` are unitary matrices.

`[ ___ ] = fixed.jacobiSVD( ___ ,numberOfSweeps)` performs `numberOfSweeps` Jacobi iterations. If `numberOfSweeps` is not supplied, the default is 10.

Examples

Compute Singular Values of Fixed-Point Matrix

Compute the singular values of a full rank scaled-double matrix.

```
A = fi([1 0 1; -1 -2 0; 0 1 -1]);
```

Define fixed-point types that will never overflow. First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Then, define the integer length based on the value of the upper bound, with one additional bit for the sign, another additional bit for intermediate CORDIC growth, and one more bit for intermediate growth to compute the Jacobi rotations. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(3,3,max(abs(A(:))));
additionalBitGrowth = 3;
integerLength = ceil(log2(svdUpperBound)) + additionalBitGrowth;
wordLength = 16;
fractionLength = wordLength - integerLength;
```

Cast the matrix `A` to the signed fixed-point type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','Fixed');
A = cast(A,'like',T.A)
```

```
A =
     1     0     1
    -1    -2     0
     0     1    -1
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

Compute the singular values of the fixed-point matrix A.

```
s = fixed.jacobiSVD(A)
```

```
s = 
    2.4502
    1.6904
    0.2295

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

The singular values are returned in a column vector in decreasing order with the same data type as the input matrix A.

Find Fixed-Point Singular Value Decomposition

Find the singular value decomposition of the fixed-point matrix A.

```
m = 4;
n = 2;
rng('default');
A = 10*randn(m,n)
```

```
A = 4×2

    5.3767    3.1877
   18.3389  -13.0769
  -22.5885   -4.3359
    8.6217    3.4262
```

Define fixed-point types that will never overflow. First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values.Then, define the integer length based on the value of the upper bound, with one additional bit for the sign, another additional bit for intermediate CORDIC growth, and one more bit for intermediate growth to compute the Jacobi rotations. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max(abs(A(:))));
additionalBitGrowth = 3;
integerLength = ceil(log2(svdUpperBound)) + additionalBitGrowth;
wordLength = 16;
fractionLength = wordLength - integerLength;
```

Cast the matrix A to the signed fixed-point type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','Fixed');
A = cast(A,'like',T.A)
```

```
A =
    5.3750    3.1875
   18.3359  -13.0781
  -22.5859   -4.3359
    8.6250    3.4297

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 7
```

Find the singular value decomposition of the fixed-point matrix A.

```
[U,s,V] = fixed.jacobiSVD(A)
```

```
U =
   -0.1583    0.2714
   -0.6411   -0.7532
    0.7036   -0.5067
   -0.2615    0.3201

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14

s =
   31.0312
   14.1484

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 7

V =
   -0.9918    0.1267
    0.1267    0.9918

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14
```

Confirm the relation `A = U*diag(s)*V'`.

```
U*diag(s)*V'
```

```
ans =
    5.3593    3.1854
   18.3815  -13.0908
  -22.5638   -4.3440
    8.6213    3.4633

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 50
        FractionLength: 35
```

Control Number of Jacobi Iterations

Use the `numberOfSweeps` parameter to control the number of iterations performed.

```
m = 5;
n = 3;
rng('default');
A = 10*randn(m,n)
```

```
A = 5×3

    5.3767  -13.0769  -13.4989
   18.3389   -4.3359   30.3492
  -22.5885    3.4262    7.2540
    8.6217   35.7840   -0.6305
    3.1877   27.6944    7.1474
```

Define fixed-point types that will never overflow. First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Then, define the integer length based on the value of the upper bound, with one additional bit for the sign, another additional bit for intermediate CORDIC growth, and one more bit for intermediate growth to compute the Jacobi rotations. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max(abs(A(:))));
additionalBitGrowth = 3;
integerLength = ceil(log2(svdUpperBound)) + additionalBitGrowth;
wordLength = 16;
fractionLength = wordLength - integerLength;
```

Cast the matrix `A` to the signed fixed-point type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','Fixed');
A = cast(A,'like',T.A)
```

```
A = 
    5.3750  -13.0625  -13.5000
   18.3438   -4.3438   30.3438
  -22.5938    3.4375    7.2500
    8.6250   35.7812   -0.6250
    3.1875   27.6875    7.1562

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 5
```

Find the singular value decomposition of the fixed-point matrix `A` and calculate the relative error.

```
[U,s,V] = fixed.jacobiSVD(A);
relativeError = norm(double(U*diag(s)*V' - A))/norm(double(A))
```

```
relativeError = 0.0060
```

Recalculate the singular value decomposition using only two Jacobi iterations and compute the relative error.

```
[U,s,V] = fixed.jacobiSVD(A,1);
relativeError = norm(double(U*diag(s)*V' - A))/norm(double(A))
```

```
relativeError = 0.0707
```

Compare the relative error between the default 10 iterations and 2 iterations.

Input Arguments

A — Input matrix
matrix

Input matrix, specified as a matrix. `A` can be a signed fixed-point `fi`, a signed scaled double `fi`, `double`, or `single` data type.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

numberOfSweeps — Number of Jacobi iterations
`10` (default) | positive integer-valued scalar

Number of Jacobi iterations, specified as a positive integer-valued scalar. Increasing the number of Jacobi iterations improves the accuracy of the singular value decomposition.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

U — Left singular vectors
columns of matrix

Left singular vectors, returned as the columns of a matrix.

For fixed-point and scaled-double inputs, `U` is returned as a signed fixed-point or scaled-double `fi` with the same word length as `A` and fraction length equal to two less than the word length. One of these integer bits is used for the sign. The other integer bit allows `+1` to be represented exactly.

For floating-point input, `U` has the same data type as `A`.

s — Singular values
column vector

Singular values, returned as a column vector. The singular values are nonnegative and returned in decreasing order. The singular values `S` have the same data type as `A`.

V — Right singular vectors
columns of matrix

Right singular vectors, returned as the columns of a matrix.

For fixed-point input and scaled-double inputs, `V` is returned as a signed fixed-point or scaled-double `fi` with the same word length as `A` and fraction length equal to two less than the word length. One of these integer bits is used for the sign. The other integer bit allows `+1` to be represented exactly.

For floating-point input, `V` has the same data type as `A`. One of these integer bits is used for the sign, and the other integer bit is so that `+1` can be represented exactly.

Tips

- The `fixed.jacobiSVD` function generates an economy sized vector output of the singular value decomposition. `[U,s,V] = fixed.jacobiSVD(A)` produces a vector `s` and unitary matrices `U` and `V` such that the dimensions of `U`, `s` and `V` are the same as the dimensions of `svd` with the `"econ"` and `"vector"` flags: `[U,s,V] = svd(A,"econ","vector")`.
- The behavior of the Square Jacobi SVD HDL Optimized block is equivalent to `[U,s,V] = fixed.jacobiSVD(A)`. The `fixed.JacobiSVD` function uses the same algorithm as the Square Jacobi SVD HDL Optimized block, with the same output data types. However, small numerical differences may exist in the least significant bit between the function and the block.

Algorithms

The `fixed.jacobiSVD` function uses the two-sided Jacobi algorithm for singular value decomposition (SVD) [1][2][3]. Compared to the sequential Golub-Kahan-Reinsch algorithm for SVD [4], the Jacobi algorithm has inherent parallelism and performs better for FPGA and ASIC applications [5]. The Jacobi method is an iterative algorithm. The **numberOfSweeps** parameter determines the number of iterations performed. Most sources indicate that 10 iterations is sufficient for the Jacobi algorithm to converge.

Version History

Introduced in R2023a

References

[1] Jacobi, Carl G. J. “Über ein leichtes Verfahren die in der Theorie der Säcularstörungen vorkommenden Gleichungen numerisch aufzulösen.” *Journal fur die reine und angewandte Mathematik* 30 (1846): 51–94.

[2] Forsythe, George E., and Peter Henrici. “The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix.” *Transactions of the American Mathematical Society* 94, no. 1 (January 1960): 1–23. https://doi.org/10.1090/S0002-9947-1960-0109825-2.

[3] Shiri, Aidin and Ghader Khosroshahi. 2019. “An FPGA Implementation of Singular Value Decomposition.” *ICEE 2019:27th Iranian Conference on Electrical Engineering, Yazd, Iran, April 30–May 2, 2019*, 416–22. IEEE. https://doi.org/10.1109/IranianCEE.2019.8786719.

[4] Golub, Gene H., and Charles F. Van Loan. *Matrix Computations*, 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013.

[5] Athi, Mrudula V., Seyed R. Zekavat, and Allan A. Struthers. “Real-Time Signal Processing of Massive Sensor Arrays via a Parallel Fast Converging SVD Algorithm: Latency, Throughput, and Resource Analysis.” *IEEE Sensors Journal* 16, no. 8 (January 2016): 2519–26.https://doi.org/10.1109/JSEN.2016.2517040.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

`fixed.jacobiSVD` generates efficient, purely integer C code.

See Also

`fixed.svd` | `svd` | Square Jacobi SVD HDL Optimized

Topics

"Singular Values"

fixed.qlessQR

Q-less QR decomposition

Syntax

```
R = fixed.qlessQR(A)
R = fixed.qlessQR(A,forgettingFactor)
R = fixed.qlessQR(A,[],regularizationParameter)
R = fixed.qlessQR(A,forgettingFactor,regularizationParameter)
```

Description

`R = fixed.qlessQR(A)` returns the upper-triangular `R` factor of the QR decomposition A = QR.

This is equivalent to computing

```
[~,R] = qr(A)
```

`R = fixed.qlessQR(A,forgettingFactor)` returns the upper-triangular `R` factor of the QR decomposition and multiplies `R` by the `forgettingFactor` before each row of `A` is processed.

`R = fixed.qlessQR(A,[],regularizationParameter)` returns the upper-triangular `R` factor of the QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where `A` is an m-by-n matrix and λ is the `regularizationParameter`.

`R = fixed.qlessQR(A,forgettingFactor,regularizationParameter)` returns the upper-triangular `R` factor of the QR decomposition of

$$\begin{bmatrix} \alpha^m \lambda I_n \\ \begin{bmatrix} \alpha^m & & & \\ & \alpha^{m-1} & & \\ & & \ddots & \\ & & & \alpha \end{bmatrix} A \end{bmatrix}$$

where α is the `forgettingFactor`, λ is the `regularizationParameter`, and `A` is an m-by-n matrix.

Examples

Solve a System of Equations Using Forward and Backward Substitution

This example shows how to solve the system of equations $(A'A)x = B$ using forward and backward substitution.

Specify the input variables, `A` and `B`.

```
rng default;
A = gallery('randsvd', [5,3], 1000);
b = [1; 1; 1; 1; 1];
```

Compute the upper-triangular factor, R, of A, where $A = QR$.

```
R = fixed.qlessQR(A);
```

Use forward and backward substitution to compute the value of X.

```
X = fixed.forwardSubstitute(R,b);
X(:) = fixed.backwardSubstitute(R,X)
```

```
X = 5×1
10^5 ×

   -0.9088
    2.7123
   -0.8958
         0
         0
```

This solution is equivalent to using the `fixed.qlessQRMatrixSolve` function.

```
x = fixed.qlessQRMatrixSolve(A,b)
```

```
x = 5×1
10^5 ×

   -0.9088
    2.7123
   -0.8958
         0
         0
```

Compute Upper-Triangular Matrix Factor Using Forgetting Factor

Using a forgetting factor with the `fixed.qlessQR` function is roughly equivalent to the Complex- and Real Partial-Systolic Q-less QR with Forgetting Factor blocks. These blocks process one row of the input matrix at a time and apply the forgetting factor before each row is processed. The `fixed.qlessQR` function takes in all rows of A at once, but carries out the computation in the same way as the blocks. The forgetting factor is applied before each row is processed.

Specifying a forgetting factor is useful when you want to stream an indefinite number of rows continuously, such as reading values from a sensor array continuously, without accumulating the data without bound.

Without using a forgetting factor, the accumulation is the square root of the number of rows, so 10000 rows would accumulate to $\sqrt{10000} = 100$.

```
A = ones(10000,3);
R = fixed.qlessQR(A)
```

```
R = 3×3

  100.0000  100.0000  100.0000
         0    0.0000    0.0000
         0         0    0.0000
```

To accrue with the effective height of m=16 rows, set the forgetting factor to the following.

```
m=16;
forgettingFactor = exp(-1/(2*m))
```

```
forgettingFactor = 0.9692
```

Using the forgetting factor, `fixed.qlessQR` would accumulate to about square root of 16.

```
R = fixed.qlessQR(A,forgettingFactor)
```

```
R = 3×3

    3.9377    3.9377    3.9377
         0    0.0000    0.0000
         0         0    0.0000
```

Input Arguments

A — Input matrix

matrix

Input matrix, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`forgettingFactor` — Forgetting factor

nonnegative scalar

Forgetting factor, specified as a nonnegative scalar between 0 and 1. The forgetting factor determines how much weight past data is given. The `forgettingFactor` value is multiplied by *R* before each row of `A` is processed.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`regularizationParameter` — Regularization parameter

`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

R — Upper-triangular factor
matrix

Upper-triangular factor, returned as a matrix that satisfies A = *QR*.

Version History

Introduced in R2020b

R2022a: Support for Tikhonov regularization parameter

The `fixed.qlessqr` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

A must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.backwardSubstitute` | `fixed.forwardSubstitute` | `fixed.qlessQRUpdate` | `fixed.qrAB` | `fixed.qrMatrixSolve` | `fixed.qlessQRMatrixSolve`

Topics
“Determine Fixed-Point Types for Q-less QR Decomposition”
“Compute Forgetting Factor Required for Streaming Input Data”

fixed.qlessQRMatrixSolve

Solve system of linear equations $(A'A)X = B$ for X using Q-less QR decomposition

Syntax

```
X = fixed.qlessQRMatrixSolve(A,B)
X = fixed.qlessQRMatrixSolve(A,B,outputType)
X = fixed.qlessQRMatrixSolve(A,B,outputType,forgettingFactor)
X = fixed.qlessQRMatrixSolve(A,B,outputType,[],regularizationParameter)
X = fixed.qlessQRMatrixSolve(A,B,outputType,forgettingFactor,
regularizationParameter)
```

Description

`X = fixed.qlessQRMatrixSolve(A,B)` solves the system of linear equations $(A'A)X = B$ using QR decomposition, without computing the Q value.

The result of this code is equivalent to computing

```
[~,R] = qr(A,0);
X = R\(R'\B)
```

or

```
X = (A'*A)\B
```

`X = fixed.qlessQRMatrixSolve(A,B,outputType)` returns the solution to the system of linear equations $(A'A)X = B$ as a variable with the output type specified by `outputType`.

`X = fixed.qlessQRMatrixSolve(A,B,outputType,forgettingFactor)` returns the solution to the system of linear equations, with the `forgettingFactor` multiplied by R after each row of `A` is processed.

`X = fixed.qlessQRMatrixSolve(A,B,outputType,[],regularizationParameter)` solves the matrix equation $\left(\lambda^2 I_n + A'A\right)X = B$ where λ is the `regularizationParameter`.

`X = fixed.qlessQRMatrixSolve(A,B,outputType,forgettingFactor,
regularizationParameter)` solves the matrix equation $A'_{\alpha,\lambda}A_{\alpha,\lambda}X = B$ where

$$A_{\alpha,\lambda} = \begin{bmatrix} \alpha^m \lambda I_n \\ \begin{bmatrix} \alpha^m & & & \\ & \alpha^{m-1} & & \\ & & \ddots & \\ & & & \alpha \end{bmatrix} A \end{bmatrix},$$

α is the `forgettingFactor`, λ is the `regularizationParameter`, and m is the number of rows in A.

Examples

Solve a System of Equations Using Q-Less QR Decomposition

This example shows how to solve the system of linear equations $(A'A)X = B$ using QR decomposition, without explicitly calculating the Q factor of the QR decomposition.

```
rng('default');
m = 6;
n = 3;
p = 1;
A = randn(m,n);
B = randn(n,p);
X = fixed.qlessQRMatrixSolve(A,B)
```

```
X = 3×1

    0.2991
    0.0523
    0.4182
```

The `fixed.qlessQRMatrixSolve` function is equivalent to the following code, however the `fixed.qlessQRMatrixSolve` function is more efficient and supports fixed-point data types.

```
X = (A'*A)\B
```

```
X = 3×1

    0.2991
    0.0523
    0.4182
```

Solve System of Equations Specifying an Output Data Type

This example shows how to specify an output data type to solve a system of equations with fixed-point data.

Define the data representing the system of equations. Define the matrix `A` as a zero-mean, normally distributed random matrix with a standard deviation of 1.

```
rng('default');
m = 6;
n = 3;
p = 1;
A0 = randn(m,n);
B0 = randn(n,p);
```

Specify fixed-point data types for `A` and `B` as to avoid overflow during the computation of QR.

```
T.A = fi([],1,22,16);
T.B = fi([],1,22,16);
A = cast(A0,'like',T.A)
```

```
A =
    0.5377   -0.4336    0.7254
    1.8339    0.3426   -0.0630
   -2.2589    3.5784    0.7147
    0.8622    2.7694   -0.2050
    0.3188   -1.3499   -0.1241
   -1.3077    3.0349    1.4897

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 22
        FractionLength: 16
```

```
B = cast(B0,'like',T.B)
```

```
B =
    1.4090
    1.4172
    0.6715

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 22
        FractionLength: 16
```

Specify an output data type to avoid overflow in the back-substitution.

```
T.X = fi([],1,29,12);
```

Use the `fixed.qlessQRMatrixSolve` function to compute the solution, X.

```
X = fixed.qlessQRMatrixSolve(A,B,T.X)
```

```
X =
    0.2988
    0.0522
    0.4180

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 29
        FractionLength: 12
```

Compare this result to the result of the built-in MATLAB® operations in double-precision floating-point.

```
X0 = (A0'*A0)\B0
```

```
X0 = 3×1

    0.2991
    0.0523
    0.4182
```

Input Arguments

A — Coefficient matrix

matrix

Coefficient matrix in the linear system of equations $(A'A)X = B$.

Data Types: single | double | fi
Complex Number Support: Yes

B — Input array

vector | matrix

Input vector or matrix representing B in the linear system of equations $(A'A)X = B$.

Data Types: single | double | fi
Complex Number Support: Yes

outputType — Output data type

numerictype object | numeric variable

Output data type, specified as a numerictype object or a numeric variable. If outputType is specified as a numerictype object, the output, X, will have the specified data type. If outputType is specified as a numeric variable, X will have the same data type as the numeric variable.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi | numerictype

forgettingFactor — Forgetting factor

nonnegative scalar

Forgetting factor, specified as a nonnegative scalar between 0 and 1. The forgetting factor determines how much weight past data is given. The forgettingFactor value is multiplied by the output of the QR decomposition, R after each row of A is processed.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

regularizationParameter — Regularization parameter

0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

Output Arguments

X — Solution

vector | matrix

Solution, returned as a vector or matrix. If A is an m-by-n matrix and B is an m-by-p matrix, then X is an n-by-p matrix.

Version History

Introduced in R2020b

R2022a: Support for Tikhonov regularization parameter

The `fixed.qlessqrmatrixsolve` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

A and *B* must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.backwardSubstitute` | `fixed.forwardSubstitute` | `fixed.qlessQR` | `fixed.qlessQRUpdate` | `fixed.qrAB` | `fixed.qrMatrixSolve`

Topics

“Algorithms to Determine Fixed-Point Types for Complex Q-less QR Matrix Solve A'AX=B”
“Determine Fixed-Point Types for Complex Q-less QR Matrix Solve A'AX=B”
“Algorithms to Determine Fixed-Point Types for Real Q-less QR Matrix Solve A'AX=B”
“Determine Fixed-Point Types for Real Q-less QR Matrix Solve A'AX=B”
“Compute Forgetting Factor Required for Streaming Input Data”

fixed.qlessqrFixedpointTypes

Determine fixed-point types for transforming A to R in-place, where R is upper-triangular factor of QR decomposition of A, without computing Q

Syntax

```
T = fixed.qlessqrFixedpointTypes(m,max_abs_A,precisionBits)
T = fixed.qlessqrFixedpointTypes(m,max_abs_A,precisionBits,
regularizationParameter)
T = fixed.qlessqrFixedpointTypes( ___ ,maxWordLength)
```

Description

`T = fixed.qlessqrFixedpointTypes(m,max_abs_A,precisionBits)` computes fixed-point types for transforming A to R in-place, where R is the upper-triangular factor of the QR decomposition of A, without computing Q. `T` is returned as a struct with field `T.A` containing a `fi` object that specifies the fixed-point type for A, which guarantees no overflow will occur in the QR algorithm.

The QR algorithm transforms A in-place into upper-triangular R, where $QR=A$ is the QR decomposition of A.

`T = fixed.qlessqrFixedpointTypes(m,max_abs_A,precisionBits, regularizationParameter)` computes fixed-point types for transforming $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ in-place to $R = Q'\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the `regularizationParameter`, QR is the economy size QR decomposition of $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$, A is an m-by-n matrix, and I_n = `eye(n)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.qlessqrFixedpointTypes( ___ ,maxWordLength)` specifies the maximum word length of the fixed-point types. `maxWordLength` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Determine Fixed-Point Types for Q-less QR Decomposition

This example shows how to use `fixed.qlessqrFixedpointTypes` to analytically determine a fixed-point type for the computation of the Q-less QR decomposition.

Define Matrix Dimensions

Specify the number of rows and columns in matrix A.

```
m = 10; % Number of rows in matrix A
n = 3;  % Number of columns in matrix A
```

Generate Matrix A

Use the helper function `realUniformRandomArray` to generate a random matrix A such that the elements of A are between -1 and $+1$.

```
rng('default')
A = fixed.example.realUniformRandomArray(-1,1,m,n);
```

Select Fixed-Point Type

Use the `fixed.qlessqrFixedpointTypes` function to select the fixed-point data type for matrix A that guarantees no overflow will occur in the transformation of A in-place to $R = Q'A$.

```
max_abs_A = 1;  % Upper bound on max(abs(A(:))
precisionBits = 24;  % Number of bits of precision
T = fixed.qlessqrFixedpointTypes(m,max_abs_A,precisionBits)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to $R = Q'A$ in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 29
        FractionLength: 24
```

Use the Specified Type to Compute the Q-less QR Decomposition

Cast the input to the type determined by `fixed.qlessqrFixedpointTypes.`

```
A = cast(A,'like',T.A);
```

Accelerate `fixed.qlessQR` by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qlessQR -args {A} -o qlessQR_mex
```

Compute the QR decomposition.

```
R = qlessQR_mex(A);
```

Verify that R is Upper-Triangular

R is an upper-triangular matrix.

```
R
```

```
R =
    2.2180    0.8559   -0.5607
         0    2.0578   -0.4017
         0         0    1.7117
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 29
        FractionLength: 24
```

```
isequal(R,triu(R))
```

```
ans = logical
   1
```

Verify the Accuracy of the Output

To evaluate the accuracy of the `fixed.qlessQR` function, compute the relative error.

$R = Q'A$, and Q is orthogonal, so $R'R = A'QQ'A = A'A$, within rounding error.

```
relative_error = norm(double(R'*R - A'*A))/norm(double(A'*A))
```

```
relative_error = 9.3865e-07
```

Suppress `mlint` warnings.

```
%#ok<*NOPTS>
```

Input Arguments

`m` — Number of rows in *A*
positive integer-valued scalar

Number of rows in A, specified as a positive integer-valued scalar.

Data Types: `double`

`max_abs_A` — Maximum of absolute value of *A*
scalar

Maximum of the absolute value of A, specified as a scalar.

Example: `max(abs(A(:)))`

Data Types: `double`

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision, specified as a positive integer-valued scalar.

Data Types: `double`

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`maxWordLength` — Maximum word length of fixed-point types
`128` (default) | positive integer

Maximum word length of fixed-point types, specified as a positive integer.

If the word length of the fixed-point type exceeds the specified maximum word length, the default of `128` bits is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`T` — Fixed-point type for *A*
struct

Fixed-point type for A, returned as a struct. The struct `T` has field `T.A` that contains a `fi` object that specifies a fixed-point type for A that guarantees no overflow will occur in the QR algorithm.

Tips

Use `fixed.qlessqrFixedpointTypes` to compute fixed-point types for the inputs of these functions and blocks.

- `fixed.qlessQR`
- Complex Burst Q-less QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition
- Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor
- Real Burst Q-less QR Decomposition
- Real Partial-Systolic Q-less QR Decomposition
- Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor

Algorithms

The number of integer bits required to prevent overflow is derived from the following bound on the growth of R [1]. The required number of integer bits is added to the number of bits of precision, `precisionBits`, of the input, plus one for the sign bit, plus one bit for intermediate CORDIC gain of approximately 1.6468 [2].

The elements of R are bounded in magnitude by

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

Version History

Introduced in R2021b

R2022b: Support for maximum word length

You can now use the `maxWordLenth` parameter to specify the maximum word length of the fixed-point types.

R2022a: Support for Tikhonov regularization parameter

The `fixed.qlessqrfixedpointtypes` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

References

[1] “Perform QR Factorization Using CORDIC”

[2] Voler, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers* EC-8 (1959): 330-334.

See Also

Functions
`fixed.qlessQR`

Blocks
Complex Burst Q-less QR Decomposition | Complex Partial-Systolic Q-less QR Decomposition | Complex Partial-Systolic Q-less QR Decomposition with Forgetting Factor | Real Burst Q-less QR Decomposition | Real Partial-Systolic Q-less QR Decomposition | Real Partial-Systolic Q-less QR Decomposition with Forgetting Factor

fixed.qlessQRUpdate

Update QR factorization

Syntax

```
R = fixed.qlessQRUpdate(R, y)
R = fixed.qlessQRUpdate(R, y, forgettingFactor)
```

Description

`R = fixed.qlessQRUpdate(R, y)` updates upper-triangular R with vector y.

This syntax is equivalent to

```
[~,R] = qr([R;y],0);
```

`R = fixed.qlessQRUpdate(R, y, forgettingFactor)` updates upper-triangular `R` with vector `y` and multiplies the result by the value specified by `forgettingFactor`.

This syntax is equivalent to

```
[~,R] = qr([R;y],0);
R(:) = forgettingFactor * R;
```

Examples

Update the Upper-Triangular Factor of a Matrix

This example shows how to update the upper-triangular factor of a matrix as new data streams in.

Define a matrix and compute the upper-triangular factor, `R`, using the `fixed.qlessQR` function.

```
rng('default');
m = 20;
n = 4;
A = randn(m,n)
```

```
A = 20×4

    0.5377    0.6715   -0.1022   -1.0891
    1.8339   -1.2075   -0.2414    0.0326
   -2.2588    0.7172    0.3192    0.5525
    0.8622    1.6302    0.3129    1.1006
    0.3188    0.4889   -0.8649    1.5442
   -1.3077    1.0347   -0.0301    0.0859
   -0.4336    0.7269   -0.1649   -1.4916
    0.3426   -0.3034    0.6277   -0.7423
    3.5784    0.2939    1.0933   -1.0616
    2.7694   -0.7873    1.1093    2.3505
      ⋮
```

```
R = fixed.qlessQR(A)
```

```
R = 4×4

    7.1017   -2.0103    1.1646    0.7999
         0    4.8784    0.5745   -0.3222
         0         0    3.1658   -0.4570
         0         0         0    4.4965
```

As new data arrives, for example new values from a sensor array, you can use the `fixed.qlessQRUpdate` function to update the upper-triangular factor with the new data.

```
y1 = [1,1,1,1];
R = fixed.qlessQRUpdate(R,y1)
```

```
R = 4×4

    7.1718   -1.8513    1.2927    0.9315
         0    5.0412    0.7646   -0.0904
         0         0    3.2332   -0.2584
         0         0         0    4.6074
```

```
y2 = [1,1,1,1];
R = fixed.qlessQRUpdate(R,y2)
```

```
R = 4×4

    7.2411   -1.6954    1.4184    1.0607
         0    5.1929    0.9371    0.1191
         0         0    3.2892   -0.0962
         0         0         0    4.6928
```

The result of updating the upper-triangular factor as new data arrives is equivalent to computing the upper-triangular factor with all of the data.

```
R = fixed.qlessQR([A;y1;y2])
```

```
R = 4×4

    7.2411   -1.6954    1.4184    1.0607
         0    5.1929    0.9371    0.1191
         0         0    3.2892   -0.0962
         0         0         0    4.6928
```

When you want to stream an indefinite number of rows continuously, such as reading values from a sensor array continuously, without accumulating the data without bound, specify a forgetting factor.

```
forgettingFactor = exp(-1/(2*m))
```

```
forgettingFactor = 0.9753
```

```
y3 = [1, 1, 1, 1];
R = fixed.qlessQRUpdate(R,y3,forgettingFactor)
```

```
R = 4×4
```

```
    7.1294   -1.5046    1.5038    1.1582
         0    5.2031    1.0676    0.3020
         0         0    3.2543    0.0379
         0         0         0    4.6431
```

Input Arguments

R — Upper-triangular input matrix
matrix

Upper triangular input, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

y — Measurement vector
vector

Measurement input, specified as a vector.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`forgettingFactor` — Forgetting factor
nonnegative scalar

Forgetting factor, specified as a nonnegative scalar between 0 and 1. The forgetting factor determines how much weight past data is given. The `forgettingFactor` value is multiplied by *R* after each row of R is processed.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

R — Updated upper-triangular matrix
matrix

Updated upper-triangular factor, returned as a matrix.

Version History
Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

R and *y* must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.backwardSubstitute` | `fixed.forwardSubstitute` | `fixed.qlessQR` | `fixed.qrAB` | `fixed.qrMatrixSolve` | `fixed.qlessQRMatrixSolve`

Topics

"Compute Forgetting Factor Required for Streaming Input Data"

fixed.qrAB

Compute $C = Q'B$ and upper-triangular factor R

Syntax

```
[C,R] = fixed.qrAB(A,B)
[C,R] = fixed.qrAB(A,B,regularizationParameter)
```

Description

`[C,R] = fixed.qrAB(A,B)` computes $C = Q'B$ and upper-triangular factor R. The function simultaneously performs Givens rotations to `A` and `B` to transform `A` into `R` and `B` into `C`.

This syntax is equivalent to

```
[C,R] = qr(A,B)
```

`[C,R] = fixed.qrAB(A,B,regularizationParameter)` computes C and R using a regularization parameter value specified by `regularizationParameter`. When a regularization parameter is specified, the function simultaneously performs Givens rotations to transform

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} \rightarrow R$$

and

$$\begin{bmatrix} 0_{n,p} \\ B \end{bmatrix} \rightarrow C$$

where A is an m-by-n matrix, B is a m-by-p matrix, and λ is the regularization parameter.

This syntax is equivalent to

```
[Q,R] = qr([regularizationParameter*eye(n); A], 0);
C = Q'[zeros(n,p);B];
```

Examples

Compute C and R Factors

This example shows how to compute the upper-triangular factor R, and $C = Q'b$.

Define the input matrices, `A`, and `b`.

```
rng('default');
m = 6;
n = 3;
p = 1;
A = randn(m,n)
```

```
A = 6×3

    0.5377   -0.4336    0.7254
    1.8339    0.3426   -0.0631
   -2.2588    3.5784    0.7147
    0.8622    2.7694   -0.2050
    0.3188   -1.3499   -0.1241
   -1.3077    3.0349    1.4897
```

```
b = randn(m,p)
```

```
b = 6×1

    1.4090
    1.4172
    0.6715
   -1.2075
    0.7172
    1.6302
```

The `fixed.qrAB` function returns the upper-triangular factor, R, and $C = Q'b$.

```
[C, R] = fixed.qrAB(A,b)
```

```
C = 3×1

   -0.3284
    0.4055
    2.5481
```

```
R = 3×3

    3.3630   -2.8841   -1.0421
         0    4.8472    0.6885
         0         0    1.3258
```

Solve System of Linear Equations Using Regularization

This example shows how to solve a system of linear equations, $Ax = b$, by computing the upper-triangular factor R, and $C = Q'b$. A regularization parameter can improve the conditioning of least squares problems, and reduce the variance of the estimates when solving linear systems of equations.

Define input matrices, `A`, and `b`.

```
rng('default');
m = 50;
n = 5;
p = 1;
A = randn(m,n);
b = randn(m,p);
```

Use the `fixed.qrAB` function to compute the upper-triangular factor, R, and $C = Q'b$.

```
[C, R] = fixed.qrAB(A, b, 0.01)
```

```
C = 5×1

   -0.6361
    1.7663
    1.5892
   -2.0638
   -0.1327


R = 5×5

    9.0631    0.7471    0.4126   -0.3606    0.1883
         0    7.2515   -1.1145    0.6011   -0.7544
         0         0    7.6132   -0.9460   -0.7062
         0         0         0    6.3065   -2.3238
         0         0         0         0    5.9297
```

Use this result to solve $Ax = b$ using `x = R\C`. Compute `x = R\C` using the `fixed.qrMatrixSolve` function.

```
x = fixed.qrMatrixSolve(R,C)
```

```
x = 5×1

   -0.1148
    0.2944
    0.1650
   -0.3355
   -0.0224
```

Compare the result to computing `x = A\b` directly.

```
x = A\b
```

```
x = 5×1

   -0.1148
    0.2944
    0.1650
   -0.3355
   -0.0224
```

Input Arguments

A — Input coefficient matrix
matrix

Input coefficient matrix, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

B — Right-hand side matrix
matrix

Right-hand side matrix, specified as a matrix.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

C — Linear system factor
matrix

Linear system factor, returned as a matrix that satisfies $C = Q'B$.

R — Upper-triangular factor
matrix

Upper-triangular factor, returned as a matrix that satisfies $A = QR$.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

A and *B* must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.backwardSubstitute` | `fixed.forwardSubstitute` | `fixed.qlessQR` | `fixed.qlessQRUpdate` | `fixed.qrMatrixSolve` | `fixed.qlessQRMatrixSolve`

Topics

“Determine Fixed-Point Types for QR Decomposition”

fixed.qrFixedpointTypes

Determine fixed-point types for transforming A and R and B to $C=Q'B$ in-place, where $QR=A$ is QR decomposition of A

Syntax

```
T = fixed.qrFixedpointTypes(m,max_abs_A,max_abs_B,precisionBits)
T = fixed.qrFixedpointTypes(m,max_abs_A,max_abs_B,precisionBits,
regularizationParameter)
T = fixed.qrFixedpointTypes( ___ ,maxWordLength)
```

Description

`T = fixed.qrFixedpointTypes(m,max_abs_A,max_abs_B,precisionBits)` returns fixed-point types for A and B that guarantee no overflow will occur in the QR algorithm.

The QR algorithm transforms A in-place into upper-triangular R and transforms B in-place into $C=Q'B$, where $QR=A$ is the QR decomposition of A.

`T = fixed.qrFixedpointTypes(m,max_abs_A,max_abs_B,precisionBits, regularizationParameter)` returns fixed-point types for transforming $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ in-place to $R = Q'\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$ and $\begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$ in-place to $C = Q'\begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$ where λ is the `regularizationParameter`, QR is the economy size QR decomposition of $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}$, A is an m-by-n matrix, p is the number of columns in B, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.qrFixedpointTypes( ___ ,maxWordLength)` specifies the maximum word length of the fixed-point types. `maxWordLength` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Determine Fixed-Point Types for QR Decomposition

This example shows how to use `fixed.qrFixedpointTypes` to analytically determine fixed-point types for the computation of the QR decomposition.

Define Matrix Dimensions

Specify the number of rows in matrices A and B, the number of columns in matrix A, and the number of columns in matrix B. This example sets B to be the identity matrix the same size as the number of rows of A.

```
m = 10; % Number of rows in matrices A and B
n = 3;  % Number of columns in matrix A
```

Generate Matrices A and B

Use the helper function `realUniformRandomArray` to generate a random matrix A such that the elements of A are between -1 and $+1$. Matrix B is the identity matrix.

```
rng('default')
A = fixed.example.realUniformRandomArray(-1,1,m,n);
B = eye(m);
```

Select Fixed-Point Types

Use `fixed.qrFixedpointTypes` to select fixed-point data types for matrices A and B that guarantee no overflow will occur in the transformation of A in-place to $R = Q'A$ and B in-place to $C = Q'B$.

```
max_abs_A = 1;  % Upper bound on max(abs(A(:))
max_abs_B = 1;  % Upper bound on max(abs(B(:))
precisionBits = 24;  % Number of bits of precision
T = fixed.qrFixedpointTypes(m,max_abs_A,max_abs_B,precisionBits)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to $R = Q'A$ in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 29
        FractionLength: 24
```

`T.B` is the type computed for transforming B to $C = Q'B$ in-place so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 29
        FractionLength: 24
```

Use the Specified Types to Compute the QR Decomposition

Cast the inputs to the types determined by `fixed.qrFixedpointTypes.`

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate `fixed.qrAB` by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qrAB -args {A,B} -o qrAB_mex
```

Compute the QR decomposition.

```
[C,R] = qrAB_mex(A,B);
```

Extract the Economy-Size Q

The function `fixed.qrAB` transforms A to $R = Q'A$ and B to $C = Q'B$. In this example, B is the identity matrix, so $Q = C'$ is the economy-size orthogonal factor of the QR decomposition.

```
Q = C';
```

Verify that Q is Orthogonal and R is Upper-Triangular

Q is orthogonal, so $Q'Q$ is the identity matrix within rounding error.

```
I = Q'*Q
```

```
I = 
    1.0000   -0.0000   -0.0000
   -0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 62
        FractionLength: 48
```

R is an upper-triangular matrix.

```
R
```

```
R = 
    2.2180    0.8559   -0.5607
         0    2.0578   -0.4017
         0         0    1.7117

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 29
        FractionLength: 24
```

```
isequal(R,triu(R))
```

```
ans = logical
   1
```

Verify the Accuracy of the Output

To evaluate the accuracy of the `fixed.qrAB` function, compute the relative error.

```
relative_error = norm(double(Q*R - A))/norm(double(A))
```

```
relative_error = 1.5886e-06
```

Suppress `mlint` warnings.

```
%#ok<*NOPTS>
```

Input Arguments

`m` — Number of rows in *A*
positive integer-valued scalar

Number of rows in *A*, specified as a positive integer-valued scalar.

Data Types: `double`

`max_abs_A` — Maximum of absolute value of *A*
scalar

Maximum of the absolute value of *A*, specified as a scalar.

Example: `max(abs(A(:)))`

Data Types: `double`

`max_abs_B` — Maximum of absolute value of *B*
scalar

Maximum of the absolute value of *B*, specified as a scalar.

Example: `max(abs(B(:)))`

Data Types: `double`

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision, specified as a positive integer-valued scalar.

Data Types: `double`

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`maxWordLength` — Maximum word length of fixed-point types
`128` (default) | positive integer

Maximum word length of fixed-point types, specified as a positive integer.

If the word length of the fixed-point type exceeds the specified maximum word length, the default of `128` bits is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

T — Fixed-point types for *A* and *B*
struct

Fixed-point types for *A* and *B*, returned as a struct. The struct `T` has fields `T.A` and `T.B`. These fields contain `fi` objects that specify fixed-point types for *A* and *B* that guarantee no overflow will occur in the QR algorithm.

The QR algorithm transforms *A* in-place into upper-triangular *R* and transforms *B* in-place into *C=Q'B* where *QR=A* is the QR decomposition of *A*.

Tips

Use `fixed.qrFixedpointTypes` to compute fixed-point types for the inputs of these functions and blocks.

- `fixed.qrAB`
- Complex Burst QR Decomposition
- Complex Partial-Systolic QR Decomposition
- Real Burst QR Decomposition
- Real Partial-Systolic QR Decomposition

Algorithms

The number of integer bits required to prevent overflow is derived from the following bounds on the growth of *R* and *C=Q'B* [1]. The required number of integer bits is added to the number of bits of precision, `precisionBits`, of the input, plus one for the sign bit, plus one bit for intermediate CORDIC gain of approximately 1.6468 [2].

The elements of *R* are bounded in magnitude by

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

The elements of *C=Q'B* are bounded in magnitude by

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|).$$

Version History

Introduced in R2021b

R2022b: Support for maximum word length

You can now use the `maxWordLenth` parameter to specify the maximum word length of the fixed-point types.

R2022a: Support for Tikhonov regularization parameter

The `fixed.qrfixedpointtypes` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

References

[1] “Perform QR Factorization Using CORDIC”

[2] Voler, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers* EC-8 (1959): 330-334.

See Also

Functions
`fixed.qrAB`

Blocks
Complex Burst QR Decomposition | Complex Partial-Systolic QR Decomposition | Real Burst QR Decomposition | Real Partial-Systolic QR Decomposition

fixed.qrMatrixSolve

Solve system of linear equations $Ax = B$ for x using QR decomposition

Syntax

```
x = fixed.qrMatrixSolve(A,B)
x = fixed.qrMatrixSolve(A,B, outputType)
x = fixed.qrMatrixSolve(A,B,outputType,regularizationParameter)
```

Description

`x = fixed.qrMatrixSolve(A,B)` solves the system of linear equations $Ax = B$ using QR decomposition.

`x = fixed.qrMatrixSolve(A,B, outputType)` returns the solution to the system of linear equations $Ax = B$ as a variable with the output type specified by `outputType`.

`x = fixed.qrMatrixSolve(A,B,outputType,regularizationParameter)` returns the solution to the system of linear equations

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} x = \begin{bmatrix} 0_{n,\,p} \\ B \end{bmatrix}$$

where A is an m-by-n matrix, B is an m-by-p matrix, and λ is the regularization parameter.

Examples

Solve a System of Equations Using QR Decomposition

This example shows how to solve a simple system of linear equations $Ax = b$, using QR decomposition.

In this example, define A as a 5-by-3 matrix with a large condition number. To solve a system of linear equations involving ill-conditioned (large condition number) non-square matrices, you must use QR decomposition.

```
rng default;
A = gallery('randsvd', [5,3], 1000000);
b = [1; 1; 1; 1; 1];
x = fixed.qrMatrixSolve(A,b)
```

```
x = 3×1
10^4 ×

   -2.3777
    7.0686
   -2.2703
```

Compare the result of the `fixed.qrMatrixSolve` function with the result of the `mldivide` or `\` function.

```
x = A\b
```

```
x = 3×1
10^4 ×

   -2.3777
    7.0686
   -2.2703
```

Specify Regularization Parameter in an Overdetermined System

This example shows the effect of a regularization parameter when solving an overdetermined system. In this example, a quantity `y` is measured at several different values of time `t` to produce the following observations.

```
t = [0 .3 .8 1.1 1.6 2.3]';
y = [.82 .72 .63 .60 .55 .50]';
```

Model the data with a decaying exponential function

$y(t) = c_1 + c_2 e^{-t}$.

The preceding equation says that the vector `y` should be approximated by a linear combination of two other vectors. One is a constant vector containing all ones and the other is the vector with components `exp(-t)`. The unknown coefficients, c_1 and c_2, can be computed by doing a least-squares fit, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations and two unknowns, represented by a 6-by-2 matrix.

```
E = [ones(size(t)) exp(-t)]
```

```
E = 6×2

    1.0000    1.0000
    1.0000    0.7408
    1.0000    0.4493
    1.0000    0.3329
    1.0000    0.2019
    1.0000    0.1003
```

Use the `fixed.qrMatrixSolve` function to get the least-squares solution.

```
c = fixed.qrMatrixSolve(E, y)
```

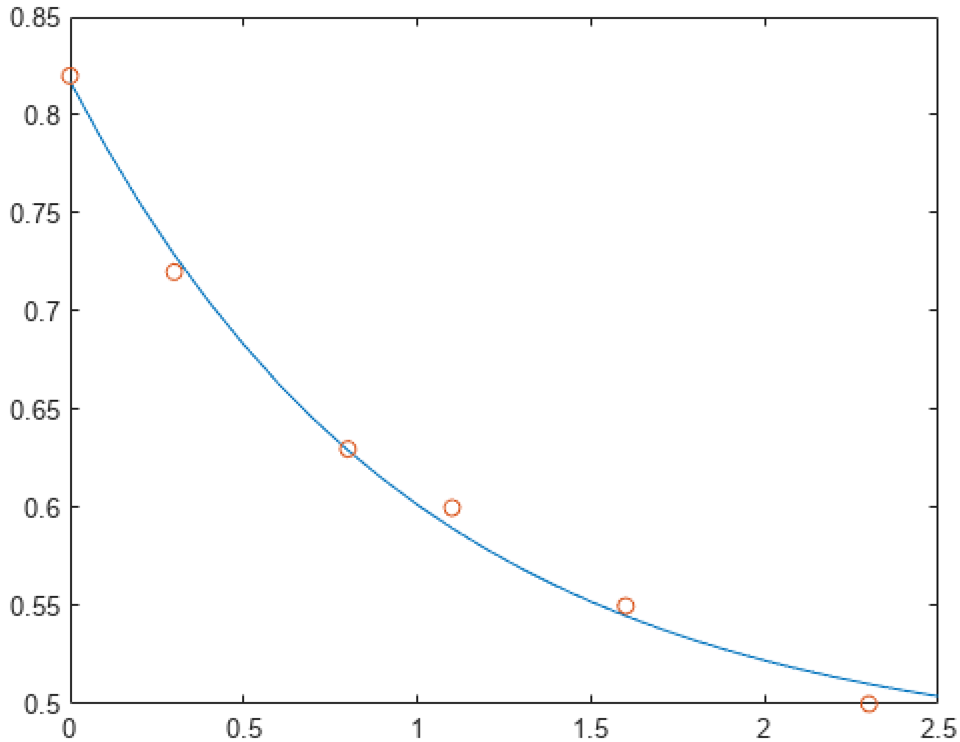
```
c = 2×1

    0.4760
    0.3413
```

In other words, the least-squares fit to the data is

$$y(t) = 0.4760 + 0.3413e^{-t}.$$

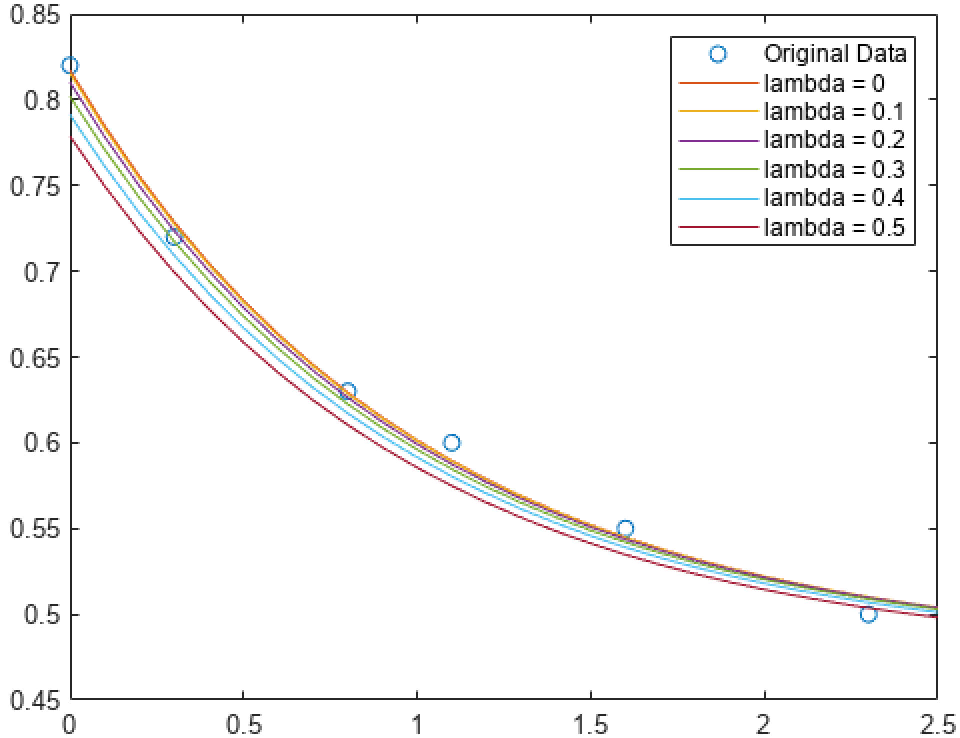
The following statements evaluate the model at regularly spaced increments in `t`, and then plot the result together with the original data:

```
T = (0:0.1:2.5)';
Y = [ones(size(T)) exp(-T)]*c;
plot(T,Y,'-',t,y,'o')
```



In cases where the input matrices are ill-conditioned, small positive values of a regularization parameter can improve the conditioning of the least squares problem, and reduce the variance of the estimates. Explore the effect of the regularization parameter on the least squares solution for this data.

```
figure;
lambda = [0:0.1:0.5];
plot(t,y,'o', 'DisplayName', 'Original Data');
for i = 1:length(lambda)
 c = fixed.qrMatrixSolve(E, y, numerictype('double'), lambda(i));
 Y = [ones(size(T)) exp(-T)]*c;
 hold on
 plot(T,Y,'-', 'DisplayName', ['lambda =', num2str(lambda(i))])
end
legend('Original Data', 'lambda = 0', 'lambda = 0.1', 'lambda = 0.2', 'lambda = 0.3', 'lambda = (
```



Input Arguments

A — Coefficient matrix
matrix

Coefficient matrix in the linear system of equations $Ax = B$.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

B — Input array
vector | matrix

Input vector or matrix representing B in the linear system of equations $Ax = B$.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`outputType` — Output data type
`numerictype` object | numeric variable

Output data type, specified as a `numerictype` object or a numeric variable. If `outputType` is specified as a `numerictype` object, the output, `x`, will have the specified data type. If `outputType` is specified as a numeric variable, `x` will have the same data type as the numeric variable.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi` | `numerictype`

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`x` — Solution
vector | matrix

Solution, returned as a vector or matrix. If `A` is an *m*-by-*n* matrix and `B` is an *m*-by-*p* matrix, then `x` is an `n`-by-`p` matrix.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Generate code for double-precision, single-precision, and fixed-point data types.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

A and *B* must be signed and use binary-point scaling. Slope-bias representation is not supported for fixed-point data types.

See Also

`fixed.backwardSubstitute` | `fixed.forwardSubstitute` | `fixed.qlessQR` | `fixed.qlessQRUpdate` | `fixed.qrAB` | `fixed.qlessQRMatrixSolve`

Topics

"Algorithms to Determine Fixed-Point Types for Complex Least-Squares Matrix Solve AX=B"
"Determine Fixed-Point Types for Complex Least-Squares Matrix Solve AX=B"
"Algorithms to Determine Fixed-Point Types for Real Least-Squares Matrix Solve AX=B"
"Determine Fixed-Point Types for Real Least-Squares Matrix Solve AX=B"

fixed.Quantizer

Quantize fixed-point numbers

Note `fixed.Quantizer` is not recommended. Use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. For more information, see Compatibility Considerations.

Description

The `fixed.Quantizer` object describes data type properties to use for quantization. After you create a `fixed.Quantizer` object, use `quantize` to quantize `fi` values.

Creation

Syntax

```
q = fixed.Quantizer
q = fixed.Quantizer(nt,rm,oa)
q = fixed.Quantizer(s,wl,fl,rm,oa)
q = fixed.Quantizer(Name,Value)
```

Description

`q = fixed.Quantizer` creates a quantizer object `q` that quantizes fixed-point numbers using the fixed-point settings of `q`.

`q = fixed.Quantizer(nt,rm,oa)` creates a fixed-point quantizer object with `numerictype` `nt`, rounding method `rm`, and overflow action `oa`.

The `numerictype`, rounding method, and overflow action apply only during the quantization. The output `q` does not have an attached `fimath`.

`q = fixed.Quantizer(s,wl,fl,rm,oa)` creates a binary-point scaled fixed-point quantizer object with signedness `s`, word length `wl`, fraction length `fl`, rounding method `rm`, and overflow action `oa`.

`q = fixed.Quantizer(Name,Value)` creates a quantizer object with the property options specified by one or more property `Name,Value` arguments.

Input Arguments

`nt` — `numerictype` object
`numerictype(true,16,15)` (default) | `numerictype` object

`numerictype` object that describes a binary-point scaled or a slope-bias scaled fixed-point data type, specified as a `numerictype` object.

If `fixed.Quantizer` uses a `numerictype` object that has either a `Signedness` of `Auto` or unspecified `Scaling`, an error occurs.

`rm` — Rounding method
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method to use for quantization, specified as one of the following:

- `'Ceiling'` — Round up to the next allowable quantized value.
- `'Convergent'` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit after rounding would be set to 0.
- `'Floor'` — Round down to the next allowable quantized value.
- `'Nearest'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.
- `'Round'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up in absolute value.
- `'Zero'` — Round negative numbers up and positive numbers down to the next allowable quantized value.

`oa` — Action to take on overflow
`'Wrap'` (default) | `'Saturate'`

Action to take on overflow, specified as one of these values:

- `'Saturate'` — Overflows saturate.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers as specified by the numeric type properties, these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- `'Wrap'` — Overflows wrap.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers as specified by the numeric type properties, these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

`s` — Whether output is signed
`1` or `true` (default) | `0` or `false`

Whether output is signed, specified as one of the following:

- `1` or `true` — Signed
- `0` or `false` — Unsigned

`wl` — Word length
16 (default) | positive scalar integer

Word length of the stored integer value of the output data in bits, specified as a positive scalar integer.

`fl` — Fraction length
15 (default) | scalar integer

Fraction length of the stored integer value of the output data in bits, specified as a scalar integer.

Properties

`Bias` — Bias
0 (default) | scalar integer

Bias associated with the quantizer object, specified as a scalar integer.

The bias is a part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. For more information, see “Fixed-point numbers” on page 4-576.

`FixedExponent` — Fixed-point exponent
-15 (default) | scalar integer

Fixed-point exponent associated with the quantizer object, specified as a scalar integer. The exponent is part of the numerical representation used to interpret a fixed-point number. The exponent of a fixed-point number is equal to the negative of the fraction length. For more information, see “Fixed-point numbers” on page 4-576.

`FractionLength` — Fraction length
15 (default) | scalar integer

Fraction length of the stored integer value of the object, in bits, specified as a scalar integer.

The fraction length automatically defaults to the best precision possible based on the value of the word length and the real-world value of the `fi` object being quantized.

`OverflowAction` — Action to take on overflow
`'Wrap'` (default) | `'Saturate'`

Action to take on overflow, specified as one of these values:

- `'Saturate'` — Overflows saturate.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers, as specified by the numeric type properties, these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- `'Wrap'` — Overflows wrap.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers, as specified by the numeric type properties, these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

Data Types: `char`

`RoundingMethod` — Rounding method
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method to use for quantization, specified as one of the following:

- `'Ceiling'` — Round up to the next allowable quantized value.
- `'Convergent'` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit after rounding would be set to 0.

- `'Floor'` — Round down to the next allowable quantized value.
- `'Nearest'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.
- `'Round'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up in absolute value.
- `'Zero'` — Round negative numbers up and positive numbers down to the next allowable quantized value.

Data Types: `char`

`Signed` — Whether output is signed
`1` or `true` (default) | `0` or `false`

Whether output is signed, specified as one of the following:

- `1` or `true` — Signed
- `0` or `false` — Unsigned

Note Although the `Signed` property is still supported, the `Signedness` property always appears in the `fixed.Quantizer` object display. If you choose to change or set the signedness of your `fixed.Quantizer` object using the `Signed` property, MATLAB updates the corresponding value of the `Signedness` property.

`Signedness` — Whether output is signed
`'Signed'` (default) | `'Unsigned'`

Whether output is signed, specified as `'Signed'` or `'Unsigned'`.

`Slope` — Slope associated with object
`2^-15` (default) | positive scalar

Slope associated with the object. The slope is part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. For more information, see “Fixed-point numbers” on page 4-576.

`SlopeAdjustmentFactor` — Slope adjustment associated with object
1 (default) | scalar greater than or equal to 1 and less than 2

Slope adjustment associated with the object, specified as a scalar greater than or equal to 1 and less than 2. The slope adjustment is equivalent to the fractional slope of a fixed-point number. The fractional slope is part of the numerical representation used to express a fixed-point number. For more information, see “Fixed-point numbers” on page 4-576.

`WordLength` — Word length
16 (default) | positive scalar integer

Word length of the stored integer value of the output data, in bits, specified as a positive scalar integer.

Object Functions

quantize   Quantize fi values using fixed.Quantizer object

Examples

Reduce Word Length Resulting From Adding Two Fixed-Point Numbers

Use `fixed.Quantizer` to reduce the word length that results from adding two fixed-point numbers.

```
q = fixed.Quantizer
x1 = fi(0.1,1,16,15);
x2 = fi(0.8,1,16,15);
y  = quantize(q,x1+x2)
```

```
q = 

  fixed.Quantizer with properties:

                   Signed: 1
               WordLength: 16
    SlopeAdjustmentFactor: 1
            FixedExponent: -15
                     Bias: 0
               Signedness: 'Signed'
                    Slope: 3.0518e-05
           FractionLength: 15
           RoundingMethod: 'Floor'
           OverflowAction: 'Wrap'


y = 

    0.9000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Quantize Binary-Point Scaled Fixed-Point `fi` to Slope-Bias Scaled Fixed-Point `fi`

Use a `fixed.Quantizer` object to change a binary-point scaled fixed-point `fi` to a slope-bias scaled fixed-point `fi`.

```
x = fi(pi,1,16,13)
q = fixed.Quantizer(numerictype(1,7,1.6,0.2),'Round','Saturate')
y = quantize(q,x)
```

```
x = 

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
            WordLength: 16
        FractionLength: 13

q =

  fixed.Quantizer with properties:

                 Signed: 1
             WordLength: 7
    SlopeAdjustmentFactor: 1.6000
          FixedExponent: 0
                   Bias: 0.2000
             Signedness: 'Signed'
                  Slope: 1.6000
         FractionLength: 0
         RoundingMethod: 'Round'
         OverflowAction: 'Saturate'


y =

    3.4000

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 7
                 Slope: 1.6
                  Bias: 0.2
```

More About

Fixed-point numbers

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

where the slope can be expressed as

$$slope = fractionalslope \times 2^{fixedexponent}$$

Tips

- Use `y = quantize(q,x)` to quantize input array `x` using the fixed-point settings of the quantizer object `q`. `x` can be any fixed-point `fi` number, except a Boolean value. If `x` is a scaled double, the `x` and `y` data will be the same, but `y` will have fixed-point settings. If `x` is a double or single, then `y = x`. This functionality lets you share the same code for both floating-point data types and fi objects when quantizers are present.
- Use `n = numerictype(q)` to get a `numerictype` for the current settings of the quantizer object `q`.
- Use `clone(q)` to create a quantizer object with the same property values as `q`.

Version History

Introduced in R2011b

R2013a: `fixed.Quantizer` is not recommended

Not recommended starting in R2013a

`fixed.Quantizer` is not recommended. Use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. There are no plans to remove `fixed.Quantizer`.

Starting in R2013a, use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. The `cast`, `zeros`, `ones`, `eye`, and `subsasgn` functions can quantize other data types in addition to `fi` objects.

Not Recommended	Recommended

`x = fi(pi,1,16,13);`
 	
`q = fixed.Quantizer(numerictype(1,7,1.6,0.2),'Round','Saturate');`
`y = quantize(q,x)`
 	
`y =`
 	
`    3.4000`
 	
`          DataTypeMode: Fixed-point: slope and bias scaling`
`            Signedness: Signed`
`            WordLength: 7`
`                 Slope: 1.6`
`                  Bias: 0.2`
	`x = fi(pi,1,16,13);`
	
	`F = fimath('RoundingMethod','Round','OverflowAction','Sa`
	`nt = fi([],1,7,1.6,0.2,F);`
	`y = cast(x,'like',nt)`
	
	`y =`
	
	`    3.4000`
	
	`          DataTypeMode: Fixed-point: slope and bias sca`
	`            Signedness: Signed`
	`            WordLength: 7`
	`                 Slope: 1.6`
	`                  Bias: 0.2`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

`fixed.Quantizer` is a handle object and must be declared as persistent in code generation.

See Also

`quantize` | `fi` | `numerictype`

fixed.realConditionNumberUpperBound

Estimate of upper bound for 2-norm condition number of real-valued matrix

Syntax

```
C = fixed.realConditionNumberUpperBound(m,n,max_abs_A,noiseStandardDeviation)
C = fixed.realConditionNumberUpperBound( ___ ,p_s)
C = fixed.realConditionNumberUpperBound( ___ ,regularizationParameter)
```

Description

`C = fixed.realConditionNumberUpperBound(m,n,max_abs_A,noiseStandardDeviation)` returns an estimate of an upper bound for the 2-norm condition number of a real-valued `m`-by-`n` matrix `A`, where `max_abs_A >= max(abs(A(:)))` and `noiseStandardDeviation` is the standard deviation of the additive random noise in `A`.

`C = fixed.realConditionNumberUpperBound( ___ ,p_s)` uses the probability `p_s` that the estimate of the lower bound of the smallest singular value is larger than the actual smallest singular value. `p_s` is an optional parameter. If not supplied or empty, then the default value is used.

`C = fixed.realConditionNumberUpperBound( ___ ,regularizationParameter)` returns an estimate of an upper bound for the 2-norm condition number of a real-valued matrix $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$, where λ is the `regularizationParameter`, A is an m-by-n matrix with m >= n, and I_n = `eye(`n`)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Condition Number of Low Rank Matrix with Additive Noise

Estimate an upper bound for the 2-norm condition number of a low rank, real-valued matrix with additive noise.

Define a real matrix `A` with additive noise.

```
m = 300;
n = 10;
rankA = 3;
A = fixed.example.realRandomLowRankMatrix(m,n,rankA);

noiseStandardDeviation = sqrt(10^(-50/10));
A = A + fixed.example.realNormalRandomArray(0,...
    noiseStandardDeviation,m,n);
```

Estimate an upper bound for the condition number of the matrix `A`.

```
C = fixed.realConditionNumberUpperBound(m,n,...
    max(abs(A(:))),noiseStandardDeviation)
```

```
C =

   1.4780e+03
```

Compare to the actual condition number of the matrix.

```
C_actual = cond(A)
```

```
C_actual =

  275.5870
```

Condition Number of Low Rank Matrix with Regularization Parameter

Estimate an upper bound for the 2-norm condition number of a low rank, real-valued matrix with additive noise, using the regularization parameter.

Define a real matrix `A` with additive noise.

```
m = 300;
n = 10;
rankA = 3;
A = fixed.example.realRandomLowRankMatrix(m,n,rankA);

noiseStandardDeviation = sqrt(10^(-50/10));
A = A + fixed.example.realNormalRandomArray(0,...
    noiseStandardDeviation,m,n);
```

Define the regularization parameter.

```
regularizationParameter = 0.01;
A = [regularizationParameter*eye(n);A];
```

Estimate an upper bound for the condition number of the matrix `A` with the regularization parameter. Use the default value for `p_s`.

```
C = fixed.realConditionNumberUpperBound(m,n,...
    max(abs(A(:))),noiseStandardDeviation,[],...
    regularizationParameter)
```

```
C =

   1.4343e+03
```

Compare to the actual condition number of the matrix.

```
C_actual = cond(A)
```

```
C_actual =

  293.4647
```

Condition Number of Full Rank Matrix

Estimate an upper bound for the 2-norm condition number of a full rank random matrix with normally distributed elements.

Define a full rank, random, real matrix `A` with normally distributed elements.

```
m = 300;
n = 10;
noiseStandardDeviation = 1;
A = fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Estimate an upper bound for the condition number of the matrix `A`.

```
C = fixed.realConditionNumberUpperBound(m,n,...
    max(abs(A(:))),noiseStandardDeviation)
```

```
C =

   19.0850
```

Compare to the actual condition number of the matrix.

```
C_actual = cond(A)
```

```
C_actual =

    1.2801
```

Input Arguments

`m` — Number of rows in matrix `A`
positive integer-valued scalar

Number of rows in matrix `A`, specified as a positive integer-valued scalar. The number of rows, `m`, must be greater than or equal to the number of columns, `n`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`n` — Number of columns in matrix `A`
positive integer-valued scalar

Number of columns in matrix `A`, specified as a positive integer-valued scalar. The number of rows, `m`, must be greater than or equal to the number of columns, `n`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`max_abs_A` — Maximum of absolute value of matrix `A`
scalar

Maximum of absolute value of matrix `A`, specified as a scalar.

Example: `max_abs_A >= max(abs(A(:)))`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`noiseStandardDeviation` — Standard deviation of additive random noise in matrix `A`
`(2^-precisionBits)/(sqrt(12))` (default) | scalar

Standard deviation of additive random noise in matrix A, specified as a scalar.

If noiseStandardDeviation is not supplied or empty, then the default value is used, which is the standard deviation of the quantization noise,

$$\sigma_q = \frac{2^{-precisionBits}}{\sqrt{12}}.$$

This value is calculated by the function fixed.realQuantizationNoiseStandardDeviation.

If noiseStandardDeviation is zero, then fixed.singularValueLowerBound will return zero for the estimate of the smallest singular value, and fixed.realConditionNumberUpperBound will return an infinite condition number.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

p_s — Probability that estimate of lower bound of smallest singular value is larger than actual smallest singular value of matrix A
2.8665e-07 (default) | scalar

Probability that estimate of lower bound of smallest singular value is larger than actual smallest singular value of matrix A, specified as a scalar.

If p_s is not supplied or empty, then the default of `p_s = (1/2)*(1+erf(-5/sqrt(2))) = 2.8665e-07` is used, which is five standard deviations below the mean. So, the probability that the estimated lower bound for the smallest singular value is less than the actual smallest singular value is `1 - p_s = 0.99999971 - p_s = 0.9999997`.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

regularizationParameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

regularizationParameter is the Tikhonov regularization parameter of the matrix `[regularizationParameter*eye(n); A]`, where A is an m-by-n matrix with m >= n.

Data Types: single | double

More About

Condition Number for Inversion

A *condition number* for a matrix and computational task measures how sensitive the answer is to changes in the input data and roundoff errors in the solution process. The *condition number for inversion* of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. The condition number for inversion gives an indication of the accuracy of the results from matrix inversion and the linear equation solution.

A large condition number indicates that a small change in the coefficient matrix A can lead to larger changes in the output b in the linear equation $Ax = b$. The extreme case is when A is so poorly conditioned that it is singular (an infinite condition number), in which case it has no inverse and the linear equation has no unique solution.

Algorithms

The condition number with respect to the inversion of matrix `A` is the ratio of the largest singular value of `A` to the smallest singular value of `A`. The `fixed.realSingularValueLowerBound` function estimates the lower bound of the smallest singular value, `s_n`, of `A`. The `fixed.singularValueUpperBound` function determines an upper bound for the largest singular value, `svdUpperBound`, of `A`. A bound on the condition number of `A` is then `cond(A) = max(svd(A))/min(svd(A)) <= svdUpperBound/s_n` [1][2][3].

Version History

Introduced in R2022b

References

[1] Bryan, Thomas A., Jenna L. Warren, Brenda Zhuang, and Jessica Clayton. Continuation in Part for "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2022.

[2] Bryan, Thomas A. and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[3] Chen, Zizhong and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no. 3 (July 2005): 603-620. https://doi.org/10.1137/040616413.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fixed.singularValueUpperBound` | `fixed.complexConditionNumberUpperBound` | `fixed.realSingularValueLowerBound` | `fixed.realQuantizationNoiseStandardDeviation` | `cond`

fixed.realQlessQRMatrixSolveFixedpointTypes

Determine fixed-point types for matrix solution of real-valued $A'AX=B$ using QR decomposition

Syntax

```
T = fixed.realQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,
precisionBits)
T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)
T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,p_s)
T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)
T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)
```

Description

`T = fixed.realQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B, precisionBits)` computes fixed-point types for the matrix solution of real-valued $A'AX=B$ using QR decomposition. T is returned as a struct with fields that specify fixed-point types for A and B that guarantee no overflow will occur in the QR algorithm transforming A in-place into upper-triangular R, where $QR=A$ is the QR decomposition of X, and X such that there is a low probability of overflow.

`T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)` specifies the standard deviation of the additive random noise in A. `noiseStandardDeviation` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,p_s)` specifies the probability that the estimate of the lower bound for the smallest singular value of A is larger than the actual smallest singular value of the matrix. `p_s` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)` computes fixed-point types for the matrix solution of real-valued

$$\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}' \cdot \begin{bmatrix}\lambda I_n \\ A\end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

where λ is the `regularizationParameter`, A is an m-by-n matrix, and I_n = `eye(n)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.realQlessQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)` specifies the maximum word length of the fixed-point types. `maxWordLength` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Algorithms to Determine Fixed-Point Types for Real Q-less QR Matrix Solve A'AX=B

This example shows the algorithms that the `fixed.realQlessQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the real matrix equation $A'AX = B$, where A is an m-by-n matrix with $m > n$, B is n-by-p, and X is n-by-p.

Overview

You can solve the fixed-point matrix equation $A'AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $R'RX = B$. To solve for X, compute $X = R\backslash(R'\backslash B)$ through forward- and backward-substitution of R into B.

You can determine appropriate fixed-point types for the matrix equation $A'AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.realQlessQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on R, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of $R = Q'A$ is

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Since computing $\mathrm{svd}(A)$ is more computationally expensive than solving the system of equations, the `fixed.realQlessQRMatrixSolveFixedpointTypes` function estimates a lower bound of $\min(\mathrm{svd}(A))$.

Fixed-point types for the solution of the matrix equation $(A'A)X = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a^2/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned}
||Av||_2 &\le ||A||_2||v||_2 \\
||Q||_2 &= 1 \\
||v||_\infty &= \max(|v(:)|) \\
||v||_\infty &\le ||v||_2 \le \sqrt{m}||v||_\infty
\end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$.

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:,j) = Q'A(:,j)$, so

$$\begin{aligned}
\max(|R(:,j)|) &= ||R(:,j)||_\infty \\
&\le ||R(:,j)||_2 \\
&= ||Q'A(:,j)||_2 \\
&\le ||Q'||_2\,||A(:,j)||_2 \\
&= ||A(:,j)||_2 \\
&\le \sqrt{m}\,||A(:,j)||_\infty \\
&= \sqrt{m}\max(|A(:,j)|) \\
&\le \sqrt{m}\max(|A(:)|).
\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Upper Bound for X = (A'A)\B

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Proof of Upper Bound for X = (A'A)\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{n}\max(|B(:)|)/\min(\mathrm{svd}(A))^2 = \infty$ and so the inequality is true.

If $A'Ax = b$ and $QR = A$ is the economy-size QR decomposition of A, then $A'Ax = R'Q'QRx = R'Rx = b$. If A is full rank then $x = R^{-1} \cdot ((R')^{-1}b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the jth column of B. Then

$$
\begin{aligned}
\max(|x(:)|) &= ||x||_\infty \\
&\le ||x||_2 \\
&= ||R^{-1}\cdot((R')^{-1}b)||_2 \\
&\le ||R^{-1}||_2\,||(R')^{-1}||_2\,||b||_2 \\
&= \left(1/\min(\mathrm{svd}(A))^2\right)\cdot ||b||_2 \\
&= ||b||_2/\min(\mathrm{svd}(A))^2 \\
&\le \sqrt{n}\,||b||_\infty/\min(\mathrm{svd}(A))^2 \\
&= \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2 .
\end{aligned}
$$

Since $\max(|x(:)|) \le \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\mathrm{svd}(A))$for real-valued A using the following formula,

$$s = \sigma_N\sqrt{2\gamma^{-1}\left(\frac{p_s\Gamma(m-n+1)\Gamma(n/2)}{2^{m-n}\Gamma\left(\frac{m+1}{2}\right)\Gamma\left(\frac{m-n+1}{2}\right)},\ \frac{m-n+1}{2}\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1-p_s$ is the probability that $s \le \min(\mathrm{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.3 from [3] and rearranging terms.

Since $s \le \min(\mathrm{svd}(A))$ with probability $1-p_s$, then you can bound the magnitude of the elements of X without computing $\mathrm{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2} \le \frac{\sqrt{n}\max(|B(:)|)}{s^2} \text{ with probability } 1-p_s.$$

You can compute s using the `fixed.realSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean, $p_s = (1+\mathrm{erf}(-5/\sqrt{2}))/2 \approx 2.8665\cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1-p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, and $X = (A'A)\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices B and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\text{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, real-valued matrices `A` and `B` are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing a real signal is $2^{-\text{precisionBits}}/\sqrt{12}$ [4,5]. Use `fixed.realQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 1.7206e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.realQlessQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.realQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 27
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = (A'A)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
        WordLength: 40
    FractionLength: 24
```

Upper Bound for R

The upper bound for R is computed using the formula $\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$, where m is the number of rows of matrix A. This upper bound is used to select a fixed-point type with the required number of bits of precision to avoid an overflow in the upper bound.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 17.3205
```

Lower Bound for min(svd(A)) for Real A

A lower bound for min(svd(A)) is estimated by the `fixed.realSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.realSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation)
```

```
estimatedSingularValueLowerBound = 0.0371
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,numSamples
    noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 17.3205
```

```
max(actualMaxR)
```

```
ans = 8.1682
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

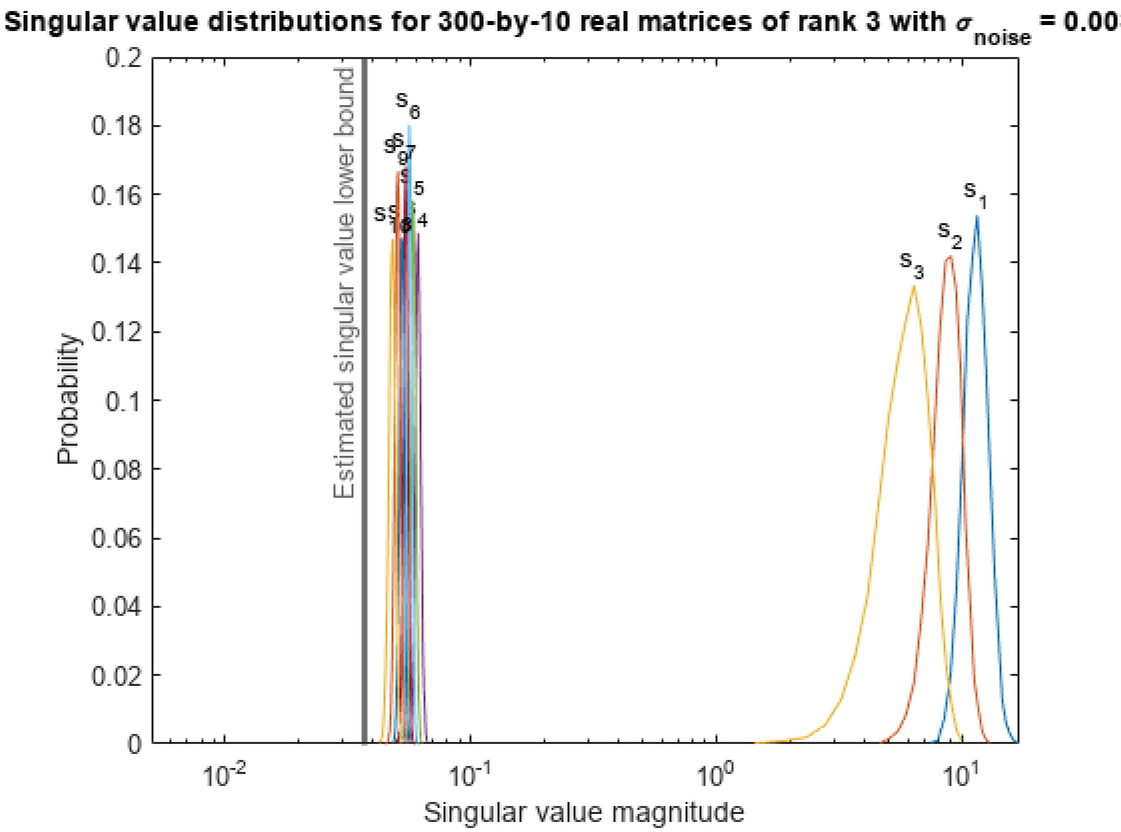
```
estimatedSingularValueLowerBound = 0.0371
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0421
```

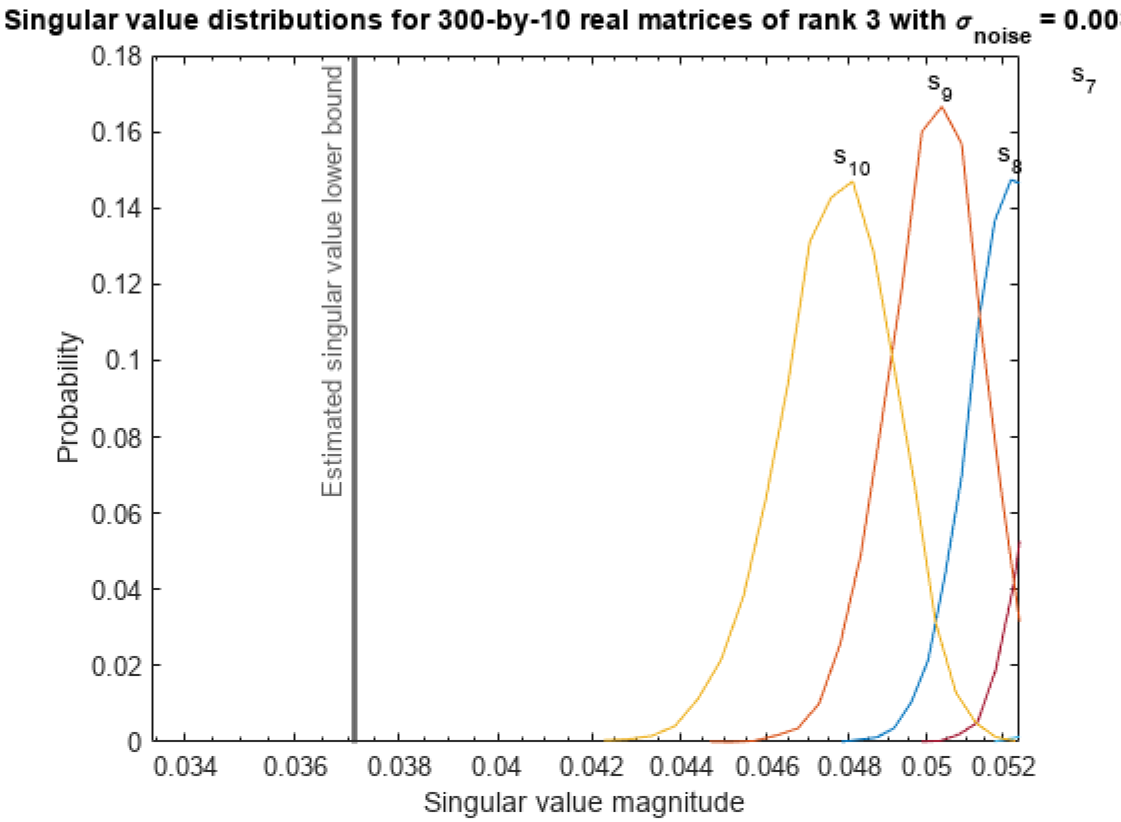
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,...
    noiseStandardDeviation,singularValues,...
    estimatedSingularValueLowerBound,"real");
```



Zoom in to the smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.realQlessQRMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDeviatio
```

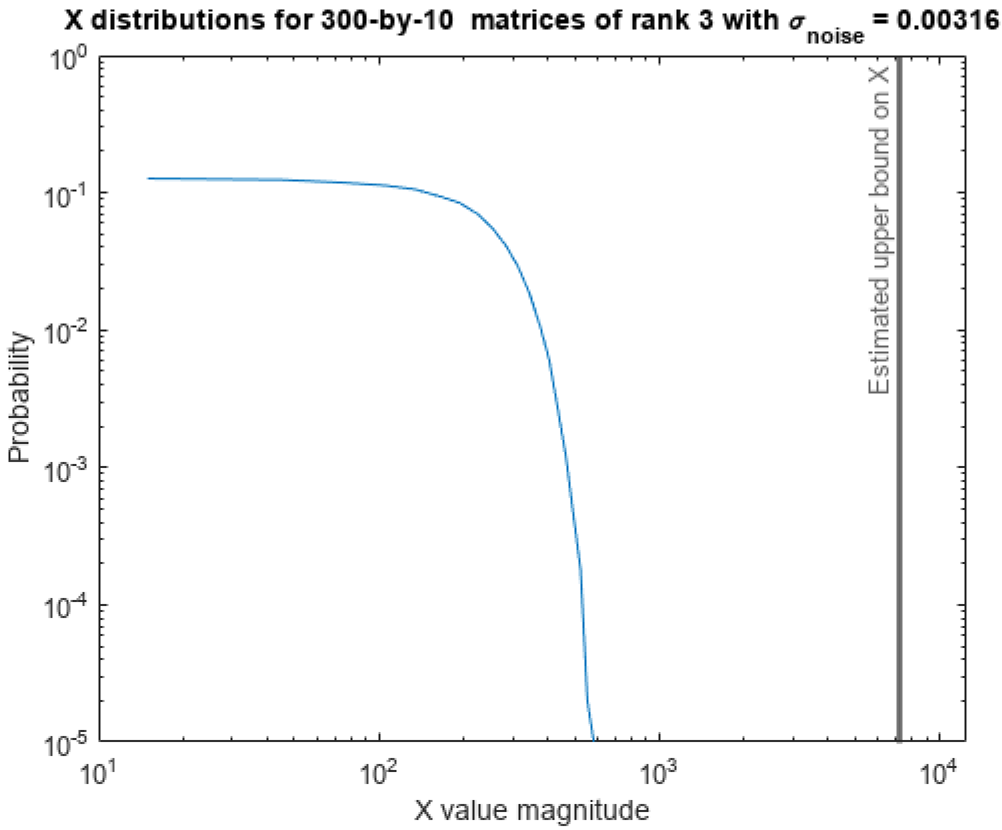
```
estimated_largest_X = 7.2565e+03
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 582.6761
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"real normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $A'AX = B$. It returns the maximum values of $R = Q'A$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,.
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = max_abs_A*fixed.example.realRandomLowRankMatrix(m,n,rankA);
        % Adding random noise makes A non-singular.
        A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.realUniformRandomArray(-max_abs_B,max_abs_B,n,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [~,R] = qr(A,0);
        X = R\(R'\B);
        actualMaxR(j) = max(abs(R(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
```

```
    end
end
```

References

1. Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.
2. Perform QR Factorization Using CORDIC. Derivation of the bound on growth when computing QR. MathWorks. 2010.
3. Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.
4. Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.
5. Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.
6. Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Real Q-less QR Matrix Solve A'AX=B

This example shows how to use the `fixed.realQlessQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the real least-squares matrix equation $A'AX = B$, where A is an m-by-n matrix with $m \geq n$, B is n-by-p, and X is n-by-p.

Fixed-point types for the solution of the matrix equation $A'AX = B$ are well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices `B` and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

p is the number of columns in matrices B and X. It corresponds to simultaneously solving a system with p right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

precisionBits defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, real-valued matrices A and B are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the upperbound function to determine the upper bounds of the fixed-point types of A and B.

max_abs_A is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

max_abs_B is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set thermalNoiseStandardDeviation to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use fixed.realQuantizationNoiseStandardDeviation to compute this. See that it is less than thermalNoiseStandardDeviation.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 1.7206e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use fixed.realQlessQRMatrixSolveFixedpointTypes to compute fixed-point types.

```
T = fixed.realQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to $R = Q'A$ in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 27
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = (A'A)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 24
```

Use the Specified Types to Solve the Matrix Equation A'AX=B

Create random matrices A and B such that rankA=rank(A). Add random measurement noise to A which will make it become full rank.

```
rng('default');
[A,B] = fixed.example.realRandomQlessQRMatrices(m,n,p,rankA);
A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.realQlessQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise [4,5].

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qlessQRMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qlessQRMatrixSolve -args {A,B,T.X} -o qlessQRMatrixSolve_mex
```

Specify output type `T.X` and compute fixed-point $X = (A'A)\backslash B$ using the QR method.

```
X = qlessQRMatrixSolve_mex(A,B,T.X);
```

Compute the relative error to verify the accuracy of the ouput.

```
relative_error = norm(double(A'*A*X - B))/norm(double(B))
```

```
relative_error = 0.0561
```

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Real Q-less QR Matrix Solve with Tikhonov Regularization

This example shows how to use the `fixed.realQlessQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the real least-squares matrix equation

$$\begin{bmatrix}\lambda I_n\\ A\end{bmatrix}^{\mathrm{T}}\begin{bmatrix}\lambda I_n\\ A\end{bmatrix}X = (\lambda^2 I_n + A^{\mathrm{T}}A)X = B$$

where A is an m-by-n matrix with $m \geq n$, B is n-by-p, X is n-by-p, $I_n = \mathrm{eye}(n)$, and λ is a regularization parameter.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices `B` and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 32;
```

Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

```
regularizationParameter = 0.01;
```

In this example, real-valued matrices `A` and `B` are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use `fixed.realQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 6.7212e-11
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use the `fixed.realQlessQRMatrixSolveFixedpointTypes` function to compute fixed-point types.

```
T = fixed.realQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation,[],regularizationParameter)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ to $R = Q^{\mathrm{T}}\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 39
        FractionLength: 32
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 35
        FractionLength: 32
```

`T.X` is the type computed for the solution $X = \left(\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}^{\mathrm{T}}\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}\right)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 48
        FractionLength: 32
```

Use the Specified Types to Solve the Matrix Equation

Create random matrices `A` and `B` such that `rankA=rank(A)`. Add random measurement noise to `A` which will make it become full rank.

```
rng('default');
[A,B] = fixed.example.realRandomQlessQRMatrices(m,n,p,rankA);
A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.realQlessQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise.

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qlessQRMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel +fixed/qlessQRMatrixSolve -args {A,B,T.X,[],regularizationParameter} -o qlessQRMatrixSolv
```

Specify output type `T.X` and compute fixed-point $X = \left(\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}^{\mathrm{T}}\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}\right)\backslash B$ using the QR method.

```
X = qlessQRMatrixSolve_mex(A,B,T.X,[],regularizationParameter);
```

Verify the Accuracy of the Output

Verify that the relative error between the fixed-point output and builtin MATLAB in double-precision floating-point is small.

$$X_{\text{double}} = \left(\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}^{\mathrm{T}}\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}\right)\backslash B$$

```
A_lambda = double([regularizationParameter*eye(n);A]);
X_double = (A_lambda'*A_lambda)\double(B);
relativeError = norm(X_double - double(X))/norm(X_double)
```

```
relativeError = 1.0133e-05
```

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Input Arguments

m — Number of rows in *A* and *B*
positive integer-valued scalar

Number of rows in *A* and *B*, specified as a positive integer-valued scalar.

Data Types: `double`

n — Number of columns in *A*
positive integer-valued scalar

Number of columns in *A*, specified as a positive integer-valued scalar.

Data Types: `double`

max_abs_A — Maximum of absolute value of *A*
scalar

Maximum of the absolute value of A, specified as a scalar.

Example: `max(abs(A(:)))`

Data Types: `double`

`max_abs_B` — Maximum of absolute value of *B*
scalar

Maximum of the absolute value of B, specified as a scalar.

Example: `max(abs(B(:)))`

Data Types: `double`

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision of the input and output, specified as a positive integer-valued scalar.

Data Types: `double`

`noiseStandardDeviation` — Standard deviation of additive random noise in *A*
scalar

Standard deviation of additive random noise in A, specified as a scalar.

If `noiseStandardDeviation` is not specified, then the default is the standard deviation of the real-valued quantization noise $\sigma_q = \left(2^{-\text{precisionBits}}\right)/(\sqrt{12})$, which is calculated by `fixed.realQuantizationNoiseStandardDeviation`.

Data Types: `double`

`p_s` — Probability that estimate of lower bound *s* is larger than actual smallest singular value of matrix
$\approx 3 \cdot 10^{-7}$ (default) | scalar

Probability that estimate of lower bound s is larger than actual smallest singular value of matrix, specified as a scalar. Use `fixed.realSingularValueLowerBound` to estimate the smallest singular value, s, of A. If `p_s` is not specified, the default value is $p_s = (1/2) \cdot (1 + \text{erf}(-5/\sqrt{2})) \approx 3 \cdot 10^{-7}$ which is 5 standard deviations below the mean, so the probability that the estimated bound for the smallest singular value is less than the actual smallest singular value is 1-$p_s \approx 0.9999997$.

Data Types: `double`

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

`regularizationParameter` is the Tikhonov regularization parameter of the matrix problem

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}' \cdot \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \left(\lambda^2 I_n + A'A\right) X = B$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`maxWordLength` — Maximum word length of fixed-point types
`128` (default) | positive integer

Maximum word length of fixed-point types, specified as a positive integer.

If the word length of the fixed-point type exceeds the specified maximum word length, the default of `128` bits is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`T` — Fixed-point types for *A*, *B*, and *X*
struct

Fixed-point types for A, B, and X, returned as a struct. The struct `T` has fields `T.A`, `T.B`, and `T.X`. These fields contain `fi` objects that specify fixed-point types for:

- A and B that guarantee no overflow will occur in the QR algorithm.

  The QR algorithm transforms A in-place into upper-triangular R, where $QR=A$ is the QR decomposition of A.
- X such that there is a low probability of overflow.

Tips

Use `fixed.realQlessQRMatrixSolveFixedpointTypes` to compute fixed-point types for the inputs of these functions and blocks.

- `fixed.qlessQRMatrixSolve`
- Real Burst Matrix Solve Using Q-less QR Decomposition
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition
- Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

Algorithms

The fixed-point type for A is computed using `fixed.qlessqrFixedpointTypes`. The required number of integer bits to prevent overflow is derived from the following bound on the growth of R [1]. The required number of integer bits is added to the number of bits of precision, `precisionBits`, of the input, plus one for the sign bit, plus one bit for intermediate CORDIC gain of approximately 1.6468 [2].

The elements of R are bounded in magnitude by

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

Matrix B is not transformed, so it does not need any additional growth bits.

The elements of X=R\(R'\B) are bounded in magnitude by

$$\max(|X(:)|) \leq \frac{n \cdot \max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Computing the singular value decomposition to derive the above bound on X is more computationally intensive than the entire matrix solve, so the `fixed.realSingularValueLowerBound` function is used to estimate a bound on `min(svd(A))`.

Version History

Introduced in R2021b

R2022b: Support for maximum word length

You can now use the `maxWordLenth` parameter to specify the maximum word length of the fixed-point types.

R2022a: Support for Tikhonov regularization parameter

The `fixed.realQlessQRMatrixSolveFixedpointTypes` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0.

References

[1] “Perform QR Factorization Using CORDIC”

[2] Voler, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers* EC-8 (1959): 330-334.

See Also

Functions
`fixed.realQuantizationNoiseStandardDeviation` | `fixed.realSingularValueLowerBound` | `fixed.qlessqrFixedpointTypes` | `fixed.qlessQRMatrixSolve`

Blocks
Real Burst Matrix Solve Using Q-less QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition | Real Partial-Systolic Matrix Solve Using Q-less QR Decomposition with Forgetting Factor

fixed.realQRMatrixSolveFixedpointTypes

Determine fixed-point types for matrix solution of real-valued *AX=B* using QR decomposition

Syntax

```
T = fixed.realQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,
precisionBits)
T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)
T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,p_s)
T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)
T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)
```

Description

`T = fixed.realQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,
precisionBits)` computes fixed-point types for the matrix solution of real-valued *AX=B* using QR decomposition. *T* is returned as a struct with fields that specify fixed-point types for *A* and *B* that guarantee no overflow will occur in the QR algorithm, and *X* such that there is a low probability of overflow.

The QR algorithm transforms *A* in-place into upper-triangular *R* and transforms *B* in-place into *C=Q'B*, where *QR=A* is the QR decomposition of *A*.

`T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,noiseStandardDeviation)` specifies the standard deviation of the additive random noise in *A*. `noiseStandardDeviation` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,p_s)` specifies the probability that the estimate of the lower bound for the smallest singular value of *A* is larger than the actual smallest singular value of the matrix. `p_s` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,regularizationParameter)` computes fixed-point types for the matrix solution of real-valued $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix}X = \begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$ where λ is the `regularizationParameter`, *A* is an *m*-by-*n* matrix, *p* is the number of columns in *B*, I_n = `eye(n)`, and $0_{n,p}$ = `zeros(n,p)`. `regularizationParameter` is an optional parameter. If not supplied or empty, then the default value is used.

`T = fixed.realQRMatrixSolveFixedpointTypes( ___ ,maxWordLength)` specifies the maximum word length of the fixed-point types. `maxWordLength` is an optional parameter. If not supplied or empty, then the default value is used.

Examples

Algorithms to Determine Fixed-Point Types for Real Least-Squares Matrix Solve AX=B

This example shows the algorithms that the `fixed.realQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the real least-squares matrix equation $AX = B$, where A is an m-by-n matrix with $m \geq n$, B is m-by-p, and X is n-by-p.

Overview

You can solve the fixed-point least-squares matrix equation $AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, and transforms matrix B in-place to $C = Q'B$, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $RX = C$. To solve for X, compute $X = R\backslash C$ through back-substitution of R into C.

You can determine appropriate fixed-point types for the least-squares matrix equation $AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.realQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on R, $C = Q'B$, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of R is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

The upper bound for the magnitude of the elements of $C = Q'B$ is

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|).$$

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$$\max(|X(:)|) \leq \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}.$$

Since computing svd(A) is more computationally expensive than solving the system of equations, the `fixed.realQRMatrixSolveFixedpointTypes` function estimates a lower bound of min(svd(A)).

Fixed-point types for the solution of the matrix equation $AX = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned}
||Av||_2 &\leq ||A||_2||v||_2 \\
||Q||_2 &= 1 \\
||v||_\infty &= \max(|v(:)|) \\
||v||_\infty &\leq ||v||_2 \leq \sqrt{m}||v||_\infty
\end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$.

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:,j) = Q'A(:,j)$, so

$$\begin{aligned}
\max(|R(:,j)|) &= ||R(:,j)||_\infty \\
&\le ||R(:,j)||_2 \\
&= ||Q'A(:,j)||_2 \\
&\le ||Q'||_2||A(:,j)||_2 \\
&= ||A(:,j)||_2 \\
&\le \sqrt{m}||A(:,j)||_\infty \\
&= \sqrt{m}\max(|A(:,j)|) \\
&\le \sqrt{m}\max(|A(:)|).
\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Upper Bound for C = Q'B

The upper bound for the magnitude of the elements of $C = Q'B$ is

$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|)$.

Proof of Upper Bound for C = Q'B

The proof of the upper bound for $C = Q'B$ is the same as the proof of the upper bound for $R = Q'A$ by substituting C for R and B for A.

Upper Bound for X = A\B

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$\max(|X(:)|) \le \dfrac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}$.

Proof of Upper Bound for X = A\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{m}\max(|B(:)|)/\min(\mathrm{svd}(A)) = \infty$ and so the inequality is true.

If A is full rank, then $x = R^{-1}(Q'b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the jth column of B. Then

$$\begin{aligned}\max(|x(:)|) &= ||x||_\infty \\ &\le ||x||_2 \\ &= ||R^{-1}\cdot(Q'b)||_2 \\ &\le ||R^{-1}||_2\,||Q'||_2\,||b||_2 \\ &= (1/\min(\mathrm{svd}(A)))\cdot 1\cdot ||b||_2 \\ &= ||b||_2/\min(\mathrm{svd}(A)) \\ &\le \sqrt{m}\,||b||_\infty/\min(\mathrm{svd}(A)) \\ &= \sqrt{m}\max(|b(:)|)/\min(\mathrm{svd}(A)).\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{m}\max(|b(:)|)/\min(\mathrm{svd}(A))$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\mathrm{svd}(A))$for real-valued A using the following formula,

$$s = \sigma_N\sqrt{2\gamma^{-1}\left(\frac{p_s\Gamma(m-n+1)\Gamma(n/2)}{2^{m-n}\Gamma\left(\frac{m+1}{2}\right)\Gamma\left(\frac{m-n+1}{2}\right)},\ \frac{m-n+1}{2}\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1-p_s$ is the probability that $s \le \min(\mathrm{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.3 from [3] and rearranging terms.

Since $s \le \min(\mathrm{svd}(A))$ with probability $1-p_s$, then you can bound the magnitude of the elements of X without computing $\mathrm{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))} \le \frac{\sqrt{m}\max(|B(:)|)}{s} \text{ with probability } 1-p_s.$$

You can compute s using the `fixed.realSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean $p_s = (1+\mathrm{erf}(-5/\sqrt{2}))/2 \approx 2.8665\cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1-p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, $C = Q'B$, and $X = A\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrix `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\text{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, real-valued matrices `A` and `B` are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing the elements of a real signal is $2^{-\text{precisionBits}}/\sqrt{12}$ [4,5]. Use the `fixed.realQuantizationNoiseStandardDeviation` function to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 1.7206e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix *A* does not have full rank (there are fewer signals of interest than number of columns of matrix *A*), and the measured system matrix *A* has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix *A* have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.realQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.realQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming *A* to *R* in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.B` is the type computed for transforming *B* to *Q'B* in-place so that it does not overflow.

```
T.B
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = A\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
```

```
          Signedness: Signed
          WordLength: 36
      FractionLength: 24
```

Upper Bounds for R and C=Q'B

The upper bounds for R and $C = Q'B$ are computed using the following formulas, where m is the number of rows of matrices A and B.

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$$

$$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|)$$

These upper bounds are used to select a fixed-point type with the required number of bits of precision to avoid overflows.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 17.3205
```

```
upperBoundQB = sqrt(m)*max_abs_B
```

```
upperBoundQB = 17.3205
```

Lower Bound for min(svd(A)) for Real A

A lower bound for min(svd(A)) is estimated by the `fixed.realSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.realSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation)
```

```
estimatedSingularValueLowerBound = 0.0371
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,...
    numSamples,noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 17.3205
```

```
max(actualMaxR)
```

```
ans = 8.3029
```

You can see that the upper bound on $C = Q'B$ compared to the measured simulation results of the maximum value of $C = Q'B$ over all runs is also within an order of magnitude.

```
upperBoundQB
```

```
upperBoundQB = 17.3205
```

```
max(actualMaxQB)
```

```
ans = 2.5707
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

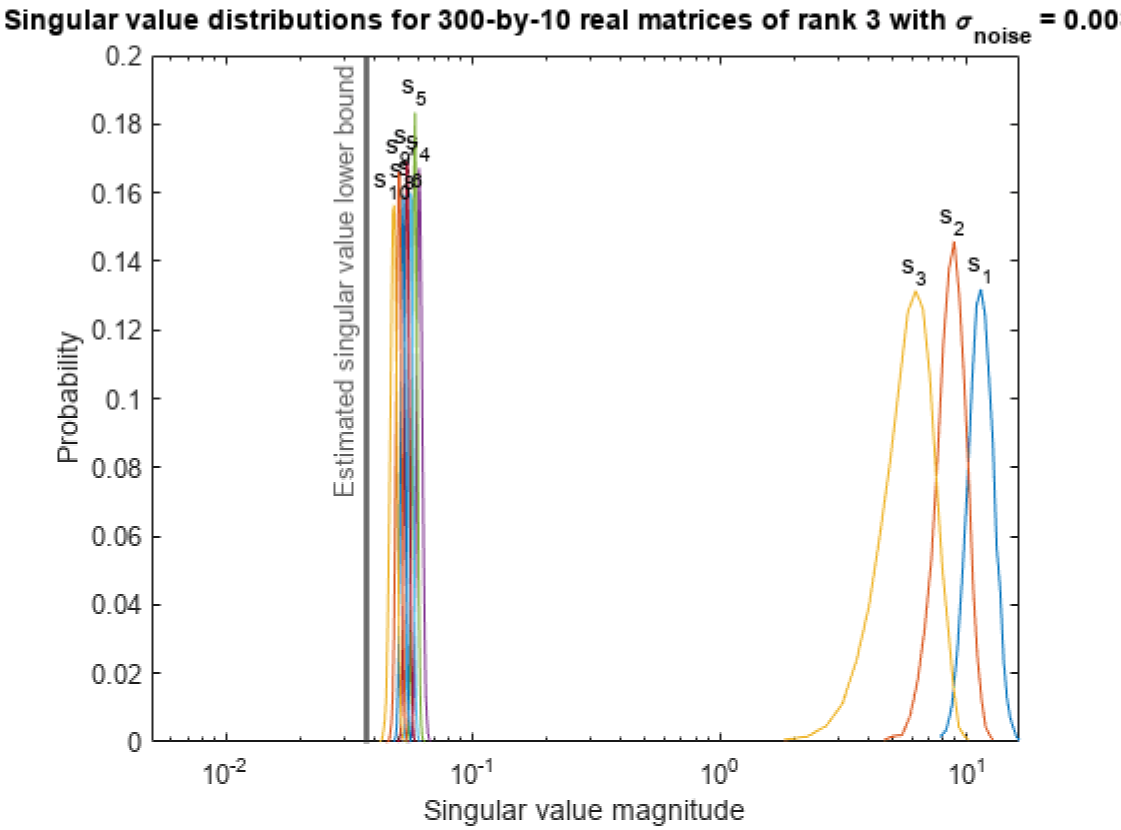
```
estimatedSingularValueLowerBound = 0.0371
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0420
```

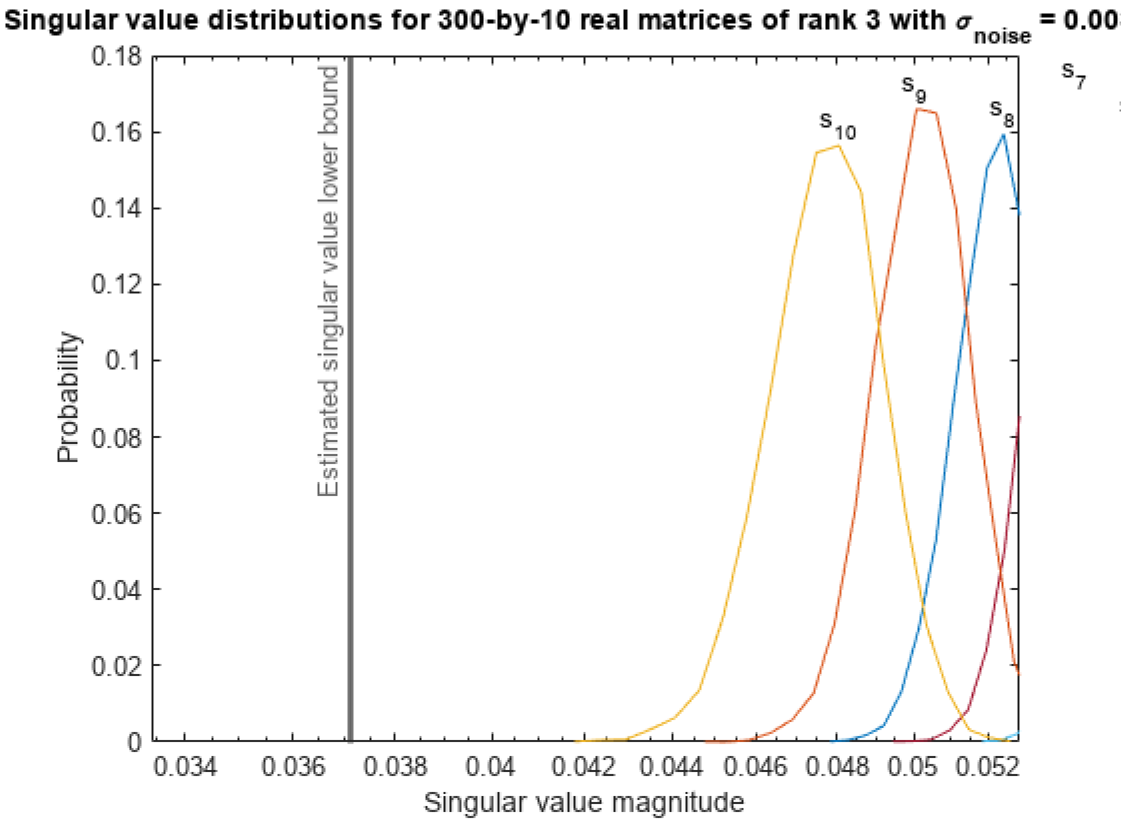
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,noiseStandardDeviation,...
    singularValues,estimatedSingularValueLowerBound,"real");
```



Zoom in to smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.realMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDeviation)
```

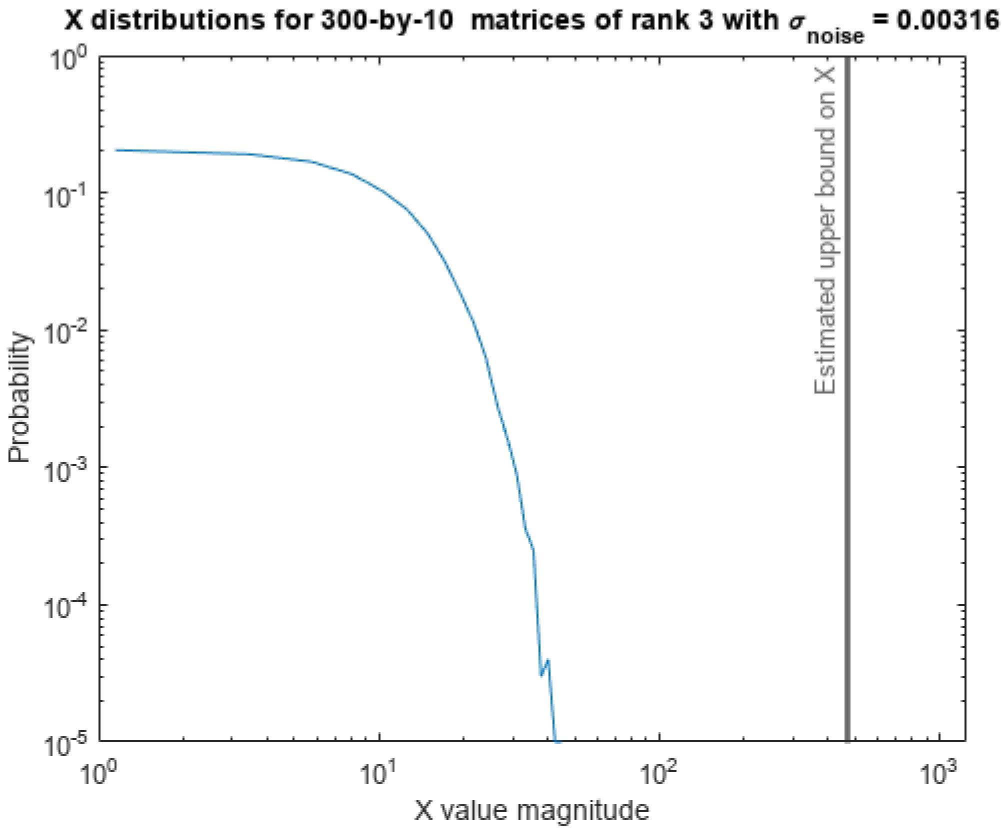
```
estimated_largest_X = 466.5772
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 44.8056
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"real normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $AX = B$. It returns the maximum values of $R = Q'A$ and $C = Q'B$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    actualMaxQB = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = max_abs_A*fixed.example.realRandomLowRankMatrix(m,n,rankA);
        % Adding normally distributed random noise makes A non-singular.
        A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.realUniformRandomArray(-max_abs_B,max_abs_B,m,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [Q,R] = qr(A,0);
        C = Q'*B;
        X = R\C;
        actualMaxR(j) = max(abs(R(:)));
```

```
        actualMaxQB(j) = max(abs(C(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
    end
end
```

References

1. Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.
2. Perform QR Factorization Using CORDIC. Derivation of the bound on growth when computing QR. MathWorks. 2010.
3. Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.
4. Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.
5. Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.
6. Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Real Least-Squares Matrix Solve AX=B

This example shows how to use the `fixed.realQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the real least-squares matrix equation $AX = B$, where A is an m-by-n matrix with $m \geq n$, B is m-by-p, and X is n-by-p.

Fixed-point types for the solution of the matrix equation $AX = B$ are well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

n is the number of columns in matrix A and rows in matrix X. In a least-squares problem, m is greater than n, and usually m is much larger than n. In a problem such as beamforming or direction finding, n corresponds to the number of sensors.

```
n = 10;
```

p is the number of columns in matrices B and X. It corresponds to simultaneously solving a system with p right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

precisionBits defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, real-valued matrices A and B are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the upperbound function to determine the upper bounds of the fixed-point types of A and B.

max_abs_A is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

max_abs_B is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set thermalNoiseStandardDeviation to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use fixed.realQuantizationNoiseStandardDeviation to compute this. See that it is less than thermalNoiseStandardDeviation.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 1.7206e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.realQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.realQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to $R = Q'A$ in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.B` is the type computed for transforming B to $C = Q'B$ in-place so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = A\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 36
        FractionLength: 24
```

Use the Specified Types to Solve the Matrix Equation AX=B

Create random matrices `A` and `B` such that `B` is in the range of `A`, and `rankA=rank(A)`. Add random measurement noise to `A` which will make it become full rank, but it will also affect the solution so that `B` is only close to the range of `A`.

```
rng('default');
[A,B] = fixed.example.realRandomLeastSquaresMatrices(m,n,p,rankA);
A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.realQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise [4,5].

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qrMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qrMatrixSolve -args {A,B,T.X} -o qrRealMatrixSolve_mex
```

Specify output type `T.X` and compute fixed-point $X = A\backslash B$ using the QR method.

```
X = qrRealMatrixSolve_mex(A,B,T.X);
```

Compute the relative error to verify the accuracy of the output.

```
relative_error = norm(double(A*X - B))/norm(double(B))
```

```
relative_error = 0.0063
```

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Determine Fixed-Point Types for Real Least-Squares Matrix Solve with Tikhonov Regularization

This example shows how to use the `fixed.realQRMatrixSolveFixedpointTypes` function to analytically determine fixed-point types for the solution of the real least-squares matrix equation

$$\begin{bmatrix} \lambda I_n \\ A \end{bmatrix} X = \begin{bmatrix} 0_{n,\,p} \\ B \end{bmatrix},$$

where A is an m-by-n matrix with $m \geq n$, B is m-by-p, X is n-by-p, $I_n = \mathrm{eye}(n)$, $0_{n,\,p} = \mathrm{zeros}(n,p)$, and λ is a regularization parameter.

The least-squares solution is

$$X_{LS} = (\lambda^2 I_n + A^{\mathrm{T}} A)^{-1} A^{\mathrm{T}} B$$

but is computed without squares or inverses.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

n is the number of columns in matrix A and rows in matrix X. In a least-squares problem, m is greater than n, and usually m is much larger than n. In a problem such as beamforming or direction finding, n corresponds to the number of sensors.

```
n = 10;
```

p is the number of columns in matrices B and X. It corresponds to simultaneously solving a system with p right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix A to be less than the number of columns. In a problem such as beamforming or direction finding, rank(A) corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

precisionBits defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 32;
```

Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

```
regularizationParameter = 0.01;
```

In this example, real-valued matrices A and B are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the upperbound function to determine the upper bounds of the fixed-point types of A and B.

max_abs_A is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

max_abs_B is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set thermalNoiseStandardDeviation to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The quantization noise standard deviation is a function of the required number of bits of precision. Use fixed.realQuantizationNoiseStandardDeviation to compute this. See that it is less than thermalNoiseStandardDeviation.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 6.7212e-11
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.realQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.realQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation,[],regularizationParameter)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming $\begin{bmatrix}\lambda I_n\\ A\end{bmatrix}$ to $R = Q^{\mathrm{T}}\begin{bmatrix}\lambda I_n\\ A\end{bmatrix}$ in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 39
        FractionLength: 32
```

`T.B` is the type computed for transforming $\begin{bmatrix}0_{n,\,p}\\ B\end{bmatrix}$ to $C = Q^{\mathrm{T}}\begin{bmatrix}0_{n,\,p}\\ B\end{bmatrix}$ in-place so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 39
        FractionLength: 32
```

`T.X` is the type computed for the solution $X = \begin{bmatrix}\lambda I_n\\ A\end{bmatrix}\backslash\begin{bmatrix}0_{n,\,p}\\ B\end{bmatrix}$, so that there is a low probability that it overflows.

```
T.X
```

```
ans =
```

```
[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 44
        FractionLength: 32
```

Use the Specified Types to Solve the Matrix Equation

Create random matrices `A` and `B` such that `B` is in the range of `A`, and `rankA=rank(A)`. Add random measurement noise to `A` which will make it become full rank, but it will also affect the solution so that `B` is only close to the range of `A`.

```
rng('default');
[A,B] = fixed.example.realRandomLeastSquaresMatrices(m,n,p,rankA);
A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
```

Cast the inputs to the types determined by `fixed.realQRMatrixSolveFixedpointTypes`. Quantizing to fixed-point is equivalent to adding random noise [4,5].

```
A = cast(A,'like',T.A);
B = cast(B,'like',T.B);
```

Accelerate the `fixed.qrMatrixSolve` function by using `fiaccel` to generate a MATLAB executable (MEX) function.

```
fiaccel fixed.qrMatrixSolve -args {A,B,T.X,regularizationParameter} -o qrRealMatrixSolve_mex
```

Specify output type `T.X` and compute fixed-point $X = A\backslash B$ using the QR method.

```
X = qrRealMatrixSolve_mex(A,B,T.X,regularizationParameter);
```

Verify the Accuracy of the Output

Verify that the relative error between the fixed-point output and the output from MATLAB using the default double-precision floating-point values is small.

$$X_{\text{double}} = \begin{bmatrix} \lambda I_n \\ A \end{bmatrix} \backslash \begin{bmatrix} 0_{n,p} \\ B \end{bmatrix}$$

```
A_lambda = double([regularizationParameter*eye(n);A]);
B_0 = [zeros(n,p);double(B)];
X_double = A_lambda\B_0;
relativeError = norm(X_double - double(X))/norm(X_double)
```

```
relativeError = 5.1152e-06
```

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Input Arguments

`m` — Number of rows in *A* and *B*
positive integer-valued scalar

Number of rows in A and B, specified as a positive integer-valued scalar.

Data Types: `double`

`n` — Number of columns in *A*
positive integer-valued scalar

Number of columns in A, specified as a positive integer-valued scalar.

Data Types: `double`

`max_abs_A` — Maximum of absolute value of *A*
scalar

Maximum of the absolute value of A, specified as a scalar.

Example: `max(abs(A(:)))`

Data Types: `double`

`max_abs_B` — Maximum of absolute value of *B*
scalar

Maximum of the absolute value of B, specified as a scalar.

Example: `max(abs(B(:)))`

Data Types: `double`

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision of the input and output, specified as a positive integer-valued scalar.

Data Types: `double`

`noiseStandardDeviation` — Standard deviation of additive random noise in *A*
scalar

Standard deviation of additive random noise in A, specified as a scalar.

If `noiseStandardDeviation` is not specified, then the default is the standard deviation of the real-valued quantization noise $\sigma_q = \left(2^{-\mathrm{precisionBits}}\right)/(\sqrt{12})$, which is calculated by `fixed.realQuantizationNoiseStandardDeviation`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`p_s` — Probability that estimate of lower bound *s* is larger than the actual smallest singular value of the matrix
≈`3·10`$^{-7}$ (default) | scalar

Probability that estimate of lower bound s is larger than the actual smallest singular value of the matrix, specified as a scalar. Use `fixed.realSingularValueLowerBound` to estimate the smallest singular value, s, of A. If `p_s` is not specified, the default value is $p_s = (1/2) \cdot (1 + \mathrm{erf}(-5/\sqrt{2})) \approx 3 \cdot 10^{-7}$ which is 5 standard deviations below the mean, so the probability that the estimated bound for the smallest singular value is less than the actual smallest singular value is $1\text{-}p_s \approx 0.9999997$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`regularizationParameter` — Regularization parameter
`0` (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

`regularizationParameter` is the Tikhonov regularization parameter of the least-squares problem $\begin{bmatrix}\lambda I_n \\ A\end{bmatrix} X = \begin{bmatrix}0_{n,p} \\ B\end{bmatrix}$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`maxWordLength` — Maximum word length of fixed-point types
`128` (default) | positive integer

Maximum word length of fixed-point types, specified as a positive integer.

If the word length of the fixed-point type exceeds the specified maximum word length, the default of `128` bits is used.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`T` — Fixed-point types for *A*, *B*, and *X*
struct

Fixed-point types for *A*, *B*, and *X*, returned as a struct. The struct `T` has fields `T.A`, `T.B`, and `T.X`. These fields contain `fi` objects that specify fixed-point types for

- *A* and *B* that guarantee no overflow will occur in the QR algorithm.

  The QR algorithm transforms *A* in-place into upper-triangular *R* and transforms *B* in-place into *C=Q'B*, where *QR=A* is the QR decomposition of *A*.
- *X* such that there is a low probability of overflow.

Tips

Use `fixed.realQRMatrixSolveFixedpointTypes` to compute fixed-point types for the inputs of these functions and blocks.

- `fixed.qrMatrixSolve`
- Real Burst Matrix Solve Using QR Decomposition

- Real Partial-Systolic Matrix Solve Using QR Decomposition

Algorithms

`T.A` and `T.B` are computed using `fixed.qrFixedpointTypes`. The number of integer bits required to prevent overflow is derived from the following bounds on the growth of R and $C=Q'B$ [1]. The required number of integer bits is added to the number of bits of precision, `precisionBits`, of the input, plus one for the sign bit, plus one bit for intermediate CORDIC gain of approximately 1.6468 [2].

The elements of R are bounded in magnitude by

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

The elements of $C=Q'B$ are bounded in magnitude by

$$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|).$$

`T.X` is computed by bounding the output, X, in the least-squares solution of $AX=B$ using the following formula [3] [4].

The elements of $X=R\backslash(Q'B)$ are bounded in magnitude by

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))}.$$

Computing the singular value decomposition to derive the above bound on X is more computationally expensive than the entire matrix solve, so the `fixed.realSingularValueLowerBound` function is used to estimate a bound on `min(svd(A))`.

Version History

Introduced in R2021b

R2022b: Support for maximum word length

You can now use the `maxWordLenth` parameter to specify the maximum word length of the fixed-point types.

R2022a: Support for Tikhonov regularization parameter

The `fixed.realQRMatrixSolveFixedpointTypes` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0.

References

[1] “Perform QR Factorization Using CORDIC”

[2] Voler, Jack E. "The CORDIC Trigonometric Computing Technique." *IRE Transactions on Electronic Computers* EC-8 (1959): 330-334.

[3] Bryan, Thomas A. and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[4] Chen, Zizhong and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no.3 (July 2005): 603-620.

See Also

Functions
`fixed.realQuantizationNoiseStandardDeviation` | `fixed.realSingularValueLowerBound` | `fixed.qrFixedpointTypes` | `fixed.qrMatrixSolve`

Blocks
Real Burst Matrix Solve Using QR Decomposition | Real Partial-Systolic Matrix Solve Using QR Decomposition

fixed.realQuantizationNoiseStandardDeviation

Estimate standard deviation of quantization noise of real-valued signal

Syntax

```
noiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(
precisionBits)
```

Description

`noiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)` returns an estimate of the quantization noise standard deviation of a real-valued signal with a quantization level $q=2^{-precisionBits}$, where `precisionBits` is the required number of bits of precision.

Examples

Estimate Standard Deviation of Quantization Noise of Real-Valued Signal

Quantizing a real signal to p bits of precision can be modeled as a linear system that adds normally distributed noise with a standard deviation of $\varsigma_{\text{noise}} = \frac{2^{-p}}{\sqrt{12}}$ [1,2].

Compute the theoretical quantization noise standard deviation with p bits of precision using the `fixed.realQuantizationNoiseStandardDeviation` function.

```
p = 14;
theoreticalQuantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(p);
```

The returned value is $\varsigma_{\text{noise}} = \frac{2^{-p}}{\sqrt{12}}$.

Create a real signal with n samples.

```
rng('default');
n = 1e6;
x = rand(1,n);
```

Quantize the signal with p bits of precision.

```
wordLength = 16;
x_quantized = quantizenumeric(x,1,wordLength,p);
```

Compute the quantization noise by taking the difference between the quantized signal and the original signal.

```
quantizationNoise = x_quantized - x;
```

Compute the measured quantization noise standard deviation.

```
measuredQuantizationNoiseStandardDeviation = std(quantizationNoise)
```

```
measuredQuantizationNoiseStandardDeviation = 1.7607e-05
```

Compare the actual quantization noise standard deviation to the theoretical and see that they are close for large values of n.

```
theoreticalQuantizationNoiseStandardDeviation
```

```
theoreticalQuantizationNoiseStandardDeviation = 1.7619e-05
```

References

1 Bernard Widrow. "A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory". In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

2 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

Input Arguments

`precisionBits` — Required number of bits of precision
positive integer-valued scalar

Required number of bits of precision, specified as a positive integer-valued scalar.

Data Types: `double`

Output Arguments

`noiseStandardDeviation` — Noise standard deviation
scalar

Noise standard deviation, returned as a scalar.

Tips

`fixed.realQuantizationNoiseStandardDeviation` is used in these functions.

- `fixed.realQRMatrixSolveFixedpointTypes`
- `fixed.realQlessQRMatrixSolveFixedpointTypes`

Algorithms

The variance of a real-valued error sequence $e(k)$ with quantization level q=$2^{-precisionBits}$ [1][2] is

$$\sigma_q^2 = \frac{1}{q}\int_{-q/2}^{q/2} e^2 de = \frac{q^2}{12} = \frac{2^{-2precisionBits}}{12}.$$

The standard deviation of a real error sequence $e(k)$ is

$$\sigma_q = \frac{2^{-precisionBits}}{\sqrt{12}}.$$

Version History

Introduced in R2021b

References

[1] Widrow, Bernard. "A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory." *IRE Transactions on Circuit Theory* 3, no.4 (December 1956): 266-276.

[2] Widrow, Bernard, and Kollár, István. *Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge, UK: Cambridge University Press, 2008.

See Also

`fixed.realQRMatrixSolveFixedpointTypes` | `fixed.realQlessQRMatrixSolveFixedpointTypes`

fixed.realSingularValueLowerBound

Estimate lower bound for smallest singular value of real-valued matrix

Syntax

```
s_n = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n)
s_n = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n,
regularizationParameter)
```

Description

`s_n = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n)` returns an estimate of a lower bound for the smallest singular value of a real-valued matrix with `m` rows and `n` columns, where $m \geq n$.

`s_n = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation,p_s_n, regularizationParameter)` returns an estimate of a lower bound for the smallest singular value of a real-valued matrix $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the `regularizationParameter`, A is an m-by-n matrix with $m >= n$, and I_n = `eye(`n`)`.

`p_s_n` and `regularizationParameter` are optional parameters. If not supplied or empty, then their default values are used.

Examples

Algorithms to Determine Fixed-Point Types for Real Q-less QR Matrix Solve A'AX=B

This example shows the algorithms that the `fixed.realQlessQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the real matrix equation $A'AX = B$, where A is an m-by-n matrix with $m > n$, B is n-by-p, and X is n-by-p.

Overview

You can solve the fixed-point matrix equation $A'AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $R'RX = B$. To solve for X, compute $X = R\backslash(R'\backslash B)$ through forward- and backward-substitution of R into B.

You can determine appropriate fixed-point types for the matrix equation $A'AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.realQlessQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on R, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of $R = Q'A$ is

$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|)$.

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Since computing $\mathrm{svd}(A)$ is more computationally expensive than solving the system of equations, the `fixed.realQlessQRMatrixSolveFixedpointTypes` function estimates a lower bound of $\min(\mathrm{svd}(A))$.

Fixed-point types for the solution of the matrix equation $(A'A)X = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a^2/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned}
||Av||_2 &\le ||A||_2\,||v||_2 \\
||Q||_2 &= 1 \\
||v||_\infty &= \max(|v(:)|) \\
||v||_\infty &\le ||v||_2 \le \sqrt{m}\,||v||_\infty
\end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\mathrm{svd}(R))} = \frac{1}{\min(\mathrm{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:, j) = Q'A(:, j)$, so

$$\begin{aligned}\max(|R(:,j)|) &= ||R(:,j)||_\infty \\ &\le ||R(:,j)||_2 \\ &= ||Q'A(:,j)||_2 \\ &\le ||Q'||_2\,||A(:,j)||_2 \\ &= ||A(:,j)||_2 \\ &\le \sqrt{m}\,||A(:,j)||_\infty \\ &= \sqrt{m}\max(|A(:,j)|) \\ &\le \sqrt{m}\max(|A(:)|)\,.\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)\,.$$

Upper Bound for X = (A'A)\B

The upper bound for the magnitude of the elements of $X = (A'A)\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Proof of Upper Bound for X = (A'A)\B

If A is not full rank, then $\min(\mathrm{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{n}\max(|B(:)|)/\min(\mathrm{svd}(A))^2 = \infty$ and so the inequality is true.

If $A'Ax = b$ and $QR = A$ is the economy-size QR decomposition of A, then $A'Ax = R'Q'QRx = R'Rx = b$. If A is full rank then $x = R^{-1} \cdot ((R')^{-1}b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the j th column of B. Then

$$\begin{aligned}\max(|x(:)|) &= ||x||_\infty \\ &\le ||x||_2 \\ &= ||R^{-1} \cdot ((R')^{-1}b)||_2 \\ &\le ||R^{-1}||_2\,||(R')^{-1}||_2\,||b||_2 \\ &= \left(1/\min(\mathrm{svd}(A))^2\right) \cdot ||b||_2 \\ &= ||b||_2/\min(\mathrm{svd}(A))^2 \\ &\le \sqrt{n}\,||b||_\infty/\min(\mathrm{svd}(A))^2 \\ &= \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2\,.\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{n}\max(|b(:)|)/\min(\mathrm{svd}(A))^2$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\mathrm{svd}(A))$for real-valued A using the following formula,

$$s = \sigma_N \sqrt{2\gamma^{-1}\left(\frac{p_s \Gamma(m-n+1)\Gamma(n/2)}{2^{m-n}\Gamma\left(\frac{m+1}{2}\right)\Gamma\left(\frac{m-n+1}{2}\right)}, \frac{m-n+1}{2}\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1 - p_s$ is the probability that $s \le \min(\mathrm{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.3 from [3] and rearranging terms.

Since $s \le \min(\mathrm{svd}(A))$ with probability $1 - p_s$, then you can bound the magnitude of the elements of X without computing $\mathrm{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{n}\max(|B(:)|)}{\min(\mathrm{svd}(A))^2} \le \frac{\sqrt{n}\max(|B(:)|)}{s^2} \text{ with probability } 1 - p_s.$$

You can compute s using the `fixed.realSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean, $p_s = (1 + \mathrm{erf}(-5/\sqrt{2}))/2 \approx 2.8665 \cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1 - p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, and $X = (A'A)\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrix `A`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrices `B` and `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\mathrm{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, real-valued matrices `A` and `B` are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and `A` and `B` are fixed-point inputs to the system, then you can use the `upperbound` function to determine the upper bounds of the fixed-point types of `A` and `B`.

`max_abs_A` is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

`max_abs_B` is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set `thermalNoiseStandardDeviation` to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing a real signal is $2^{-\text{precisionBits}}/\sqrt{12}$ [4,5]. Use `fixed.realQuantizationNoiseStandardDeviation` to compute this. See that it is less than `thermalNoiseStandardDeviation`.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 1.7206e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use `fixed.realQlessQRMatrixSolveFixedpointTypes` to compute fixed-point types.

```
T = fixed.realQlessQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

`T.A` is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.B` is the type computed for B so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 27
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = (A'A)\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 24
```

Upper Bound for R

The upper bound for R is computed using the formula $\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|)$, where m is the number of rows of matrix A. This upper bound is used to select a fixed-point type with the required number of bits of precision to avoid an overflow in the upper bound.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 17.3205
```

Lower Bound for min(svd(A)) for Real A

A lower bound for min(svd(A)) is estimated by the `fixed.realSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.realSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation)
```

```
estimatedSingularValueLowerBound = 0.0371
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,numSamples
    noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 17.3205
```

```
max(actualMaxR)
```

```
ans = 8.1682
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

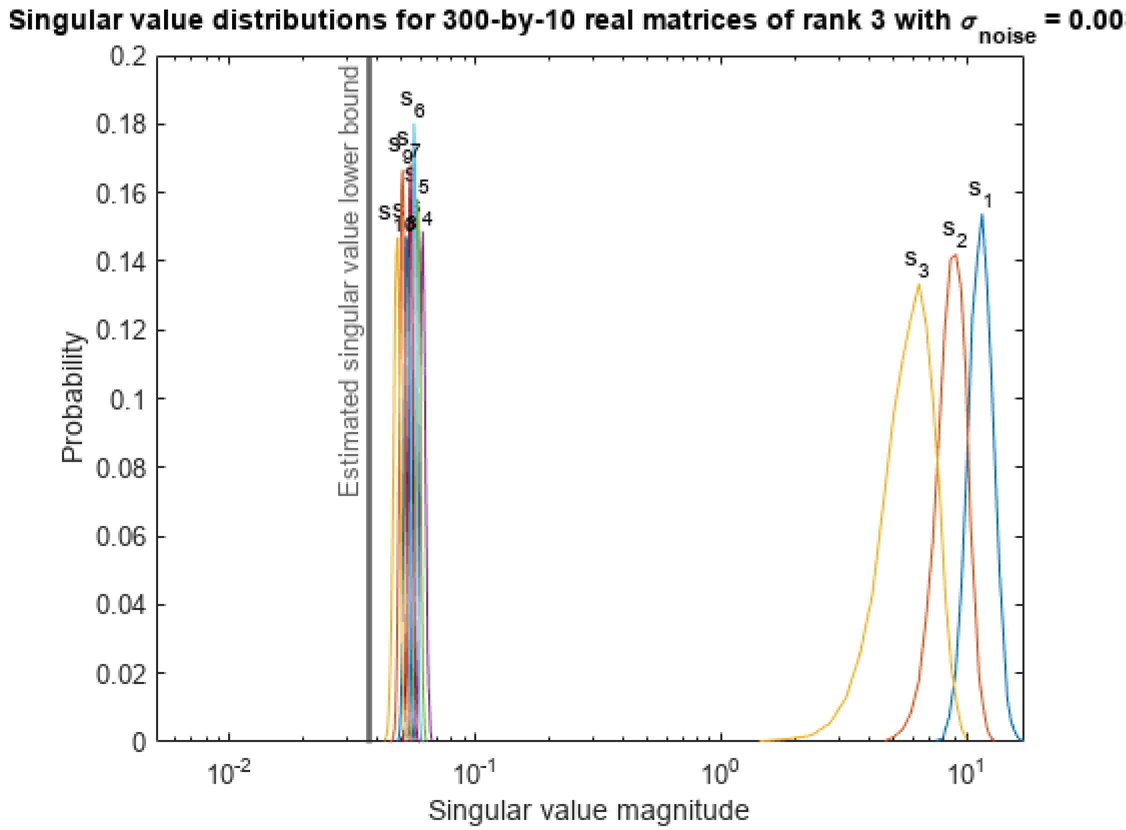
```
estimatedSingularValueLowerBound = 0.0371
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0421
```

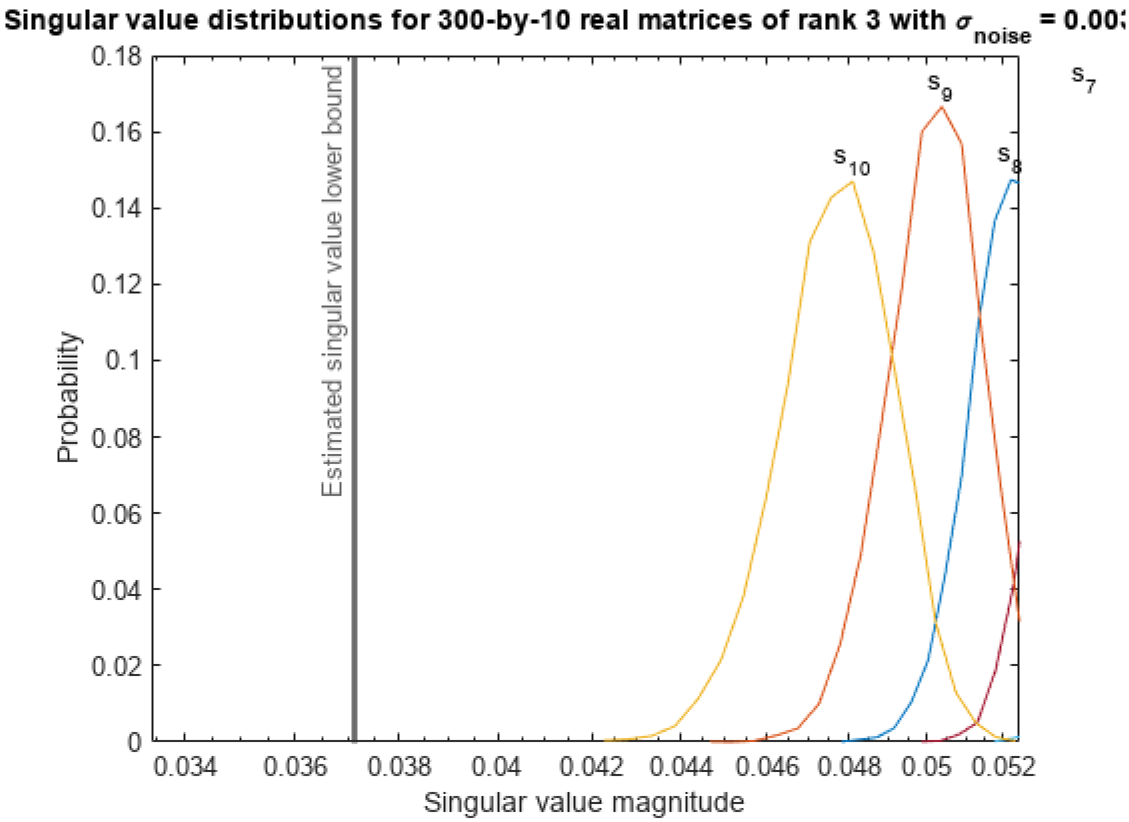
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,...
    noiseStandardDeviation,singularValues,...
    estimatedSingularValueLowerBound,"real");
```



Zoom in to the smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.realQlessQRMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDeviatio
```

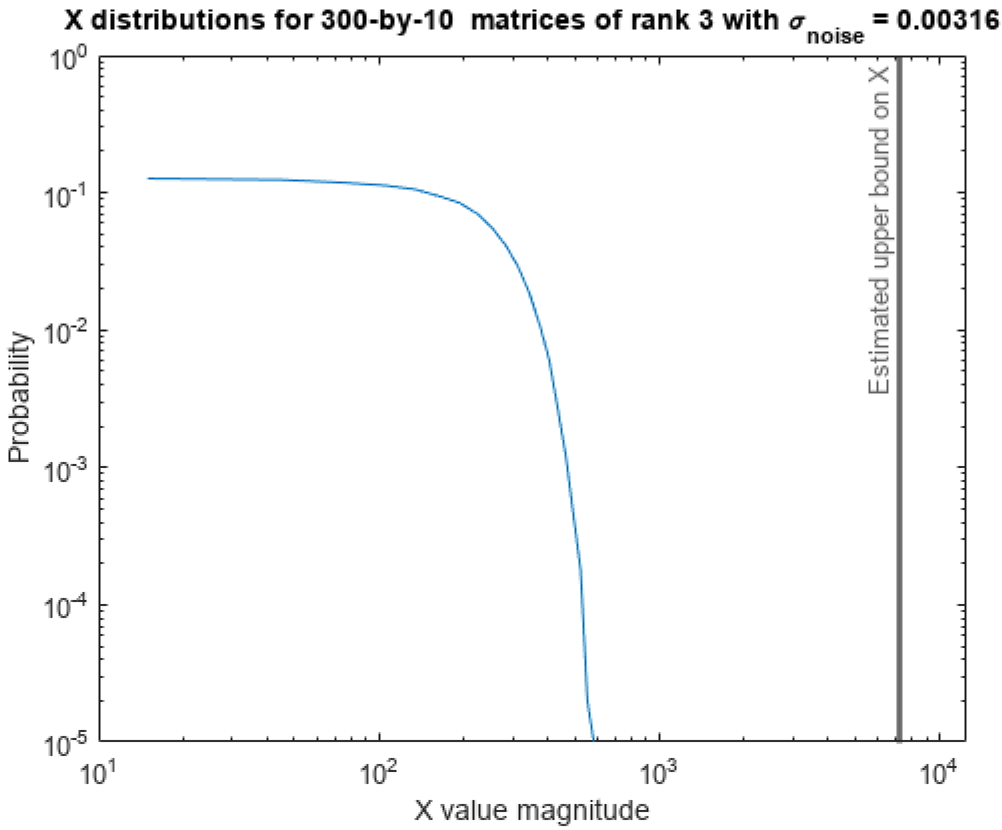
```
estimated_largest_X = 7.2565e+03
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 582.6761
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"real normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $A'AX = B$. It returns the maximum values of $R = Q'A$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,.
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = max_abs_A*fixed.example.realRandomLowRankMatrix(m,n,rankA);
        % Adding random noise makes A non-singular.
        A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.realUniformRandomArray(-max_abs_B,max_abs_B,n,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [~,R] = qr(A,0);
        X = R\(R'\B);
        actualMaxR(j) = max(abs(R(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
```

```
    end
end
```

References

1 Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.

2 Perform QR Factorization Using CORDIC. Derivation of the bound on growth when computing QR. MathWorks. 2010.

3 Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.

4 Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

5 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

6 Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress mlint warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Algorithms to Determine Fixed-Point Types for Real Least-Squares Matrix Solve AX=B

This example shows the algorithms that the `fixed.realQRMatrixSolveFixedpointTypes` function uses to analytically determine fixed-point types for the solution of the real least-squares matrix equation $AX = B$, where A is an m-by-n matrix with $m \geq n$, B is m-by-p, and X is n-by-p.

Overview

You can solve the fixed-point least-squares matrix equation $AX = B$ using QR decomposition. Using a sequence of orthogonal transformations, QR decomposition transforms matrix A in-place to upper triangular R, and transforms matrix B in-place to $C = Q'B$, where $QR = A$ is the economy-size QR decomposition. This reduces the equation to an upper-triangular system of equations $RX = C$. To solve for X, compute $X = R\backslash C$ through back-substitution of R into C.

You can determine appropriate fixed-point types for the least-squares matrix equation $AX = B$ by selecting the fraction length based on the number of bits of precision defined by your requirements. The `fixed.realQRMatrixSolveFixedpointTypes` function analytically computes the following upper bounds on R, $C = Q'B$, and X to determine the number of integer bits required to avoid overflow [1,2,3].

The upper bound for the magnitude of the elements of R is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

The upper bound for the magnitude of the elements of $C = Q'B$ is

$$\max(|C(:)|) \leq \sqrt{m}\max(|B(:)|).$$

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$$\max(|X(:)|) \leq \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))}.$$

Since computing $\text{svd}(A)$ is more computationally expensive than solving the system of equations, the `fixed.realQRMatrixSolveFixedpointTypes` function estimates a lower bound of $\min(\text{svd}(A))$.

Fixed-point types for the solution of the matrix equation $AX = B$ are generally well-bounded if the number of rows, m, of A are much greater than the number of columns, n (i.e. $m \gg n$), and A is full rank. If A is not inherently full rank, then it can be made so by adding random noise. Random noise naturally occurs in physical systems, such as thermal noise in radar or communications systems. If $m = n$, then the dynamic range of the system can be unbounded, for example in the scalar equation $x = a/b$ and $a, b \in [-1, 1]$, then x can be arbitrarily large if b is close to 0.

Proofs of the Bounds

Properties and Definitions of Vector and Matrix Norms

The proofs of the bounds use the following properties and definitions of matrix and vector norms, where Q is an orthogonal matrix, and v is a vector of length m [6].

$$\begin{aligned}
||Av||_2 &\leq ||A||_2||v||_2 \\
||Q||_2 &= 1 \\
||v||_\infty &= \max(|v(:)|) \\
||v||_\infty &\leq ||v||_2 \leq \sqrt{m}||v||_\infty
\end{aligned}$$

If A is an m-by-n matrix and $QR = A$ is the economy-size QR decomposition of A, where Q is orthogonal and m-by-n and R is upper-triangular and n-by-n, then the singular values of R are equal to the singular values of A. If A is nonsingular, then

$$||R^{-1}||_2 = ||(R')^{-1}||_2 = \frac{1}{\min(\text{svd}(R))} = \frac{1}{\min(\text{svd}(A))}$$

Upper Bound for R = Q'A

The upper bound for the magnitude of the elements of R is

$$\max(|R(:)|) \leq \sqrt{m}\max(|A(:)|).$$

Proof of Upper Bound for R = Q'A

The jth column of R is equal to $R(:, j) = Q'A(:, j)$, so

$$\begin{aligned}
\max(|R(:, j)|) &= ||R(:, j)||_\infty \\
&\leq ||R(:, j)||_2 \\
&= ||Q'A(:, j)||_2 \\
&\leq ||Q'||_2||A(:, j)||_2 \\
&= ||A(:, j)||_2 \\
&\leq \sqrt{m}||A(:, j)||_\infty \\
&= \sqrt{m}\max(|A(:, j)|) \\
&\leq \sqrt{m}\max(|A(:)|).
\end{aligned}$$

Since $\max(|R(:,j)|) \le \sqrt{m}\max(|A(:)|)$ for all $1 \le j$, then

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|).$$

Upper Bound for C = Q'B

The upper bound for the magnitude of the elements of $C = Q'B$ is

$$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|).$$

Proof of Upper Bound for C = Q'B

The proof of the upper bound for $C = Q'B$ is the same as the proof of the upper bound for $R = Q'A$ by substituting C for R and B for A.

Upper Bound for X = A\B

The upper bound for the magnitude of the elements of $X = A\backslash B$ is

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))}.$$

Proof of Upper Bound for X = A\B

If A is not full rank, then $\min(\text{svd}(A)) = 0$, and if B is not equal to zero, then $\sqrt{m}\max(|B(:)|)/\min(\text{svd}(A)) = \infty$ and so the inequality is true.

If A is full rank, then $x = R^{-1}(Q'b)$. Let $x = X(:,j)$ be the jth column of X, and $b = B(:,j)$ be the jth column of B. Then

$$\begin{aligned}
\max(|x(:)|) &= ||x||_\infty \\
&\le ||x||_2 \\
&= ||R^{-1} \cdot (Q'b)||_2 \\
&\le ||R^{-1}||_2\,||Q'||_2\,||b||_2 \\
&= (1/\min(\text{svd}(A))) \cdot 1 \cdot ||b||_2 \\
&= ||b||_2/\min(\text{svd}(A)) \\
&\le \sqrt{m}\,||b||_\infty/\min(\text{svd}(A)) \\
&= \sqrt{m}\max(|b(:)|)/\min(\text{svd}(A)).
\end{aligned}$$

Since $\max(|x(:)|) \le \sqrt{m}\max(|b(:)|)/\min(\text{svd}(A))$ for all rows and columns of B and X, then

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\text{svd}(A))}.$$

Lower Bound for min(svd(A))

You can estimate a lower bound s of $\min(\text{svd}(A))$for real-valued A using the following formula,

$$s = \sigma_N \sqrt{2\gamma^{-1}\left(\frac{p_s \Gamma(m-n+1)\Gamma(n/2)}{2^{m-n}\Gamma\left(\frac{m+1}{2}\right)\Gamma\left(\frac{m-n+1}{2}\right)},\ \frac{m-n+1}{2}\right)}$$

where σ_N is the standard deviation of random noise added to the elements of A, $1 - p_s$ is the probability that $s \le \min(\mathrm{svd}(A))$, Γ is the `gamma` function, and $\gamma^{-1}$is the inverse incomplete gamma function `gammaincinv`.

The proof is found in [1]. It is derived by integrating the formula in Lemma 3.3 from [3] and rearranging terms.

Since $s \le \min(\mathrm{svd}(A))$ with probability $1 - p_s$, then you can bound the magnitude of the elements of X without computing $\mathrm{svd}(A)$,

$$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))} \le \frac{\sqrt{m}\max(|B(:)|)}{s} \text{ with probability } 1 - p_s.$$

You can compute s using the `fixed.realSingularValueLowerBound` function which uses a default probability of 5 standard deviations below the mean $p_s = (1 + \mathrm{erf}(-5/\sqrt{2}))/2 \approx 2.8665 \cdot 10^{-7}$, so the probability that the estimated bound for the smallest singular value s is less than the actual smallest singular value of A is $1 - p_s \approx 0.9999997$.

Example

This example runs a simulation with many random matrices and compares the analytical bounds with the actual singular values of A and the actual largest elements of $R = Q'A$, $C = Q'B$, and $X = A\backslash B$.

Define System Parameters

Define the matrix attributes and system parameters for this example.

`m` is the number of rows in matrices `A` and `B`. In a problem such as beamforming or direction finding, `m` corresponds to the number of samples that are integrated over.

```
m = 300;
```

`n` is the number of columns in matrix `A` and rows in matrix `X`. In a least-squares problem, `m` is greater than `n`, and usually `m` is much larger than `n`. In a problem such as beamforming or direction finding, `n` corresponds to the number of sensors.

```
n = 10;
```

`p` is the number of columns in matrices `B` and `X`. It corresponds to simultaneously solving a system with `p` right-hand sides.

```
p = 1;
```

In this example, set the rank of matrix `A` to be less than the number of columns. In a problem such as beamforming or direction finding, $\mathrm{rank}(A)$ corresponds to the number of signals impinging on the sensor array.

```
rankA = 3;
```

`precisionBits` defines the number of bits of precision required for the matrix solve. Set this value according to system requirements.

```
precisionBits = 24;
```

In this example, real-valued matrices `A` and `B` are constructed such that the magnitude of their elements is less than or equal to one. Your own system requirements will define what those values are. If you don't know what they are, and A and B are fixed-point inputs to the system, then you can use the upperbound function to determine the upper bounds of the fixed-point types of A and B.

max_abs_A is an upper bound on the maximum magnitude element of A.

```
max_abs_A = 1;
```

max_abs_B is an upper bound on the maximum magnitude element of B.

```
max_abs_B = 1;
```

Thermal noise standard deviation is the square root of thermal noise power, which is a system parameter. A well-designed system has the quantization level lower than the thermal noise. Here, set thermalNoiseStandardDeviation to the equivalent of -50dB noise power.

```
thermalNoiseStandardDeviation = sqrt(10^(-50/10))
```

```
thermalNoiseStandardDeviation = 0.0032
```

The standard deviation of the noise from quantizing the elements of a real signal is $2^{-\text{precisionBits}}/\sqrt{12}$ [4,5]. Use the fixed.realQuantizationNoiseStandardDeviation function to compute this. See that it is less than thermalNoiseStandardDeviation.

```
quantizationNoiseStandardDeviation = fixed.realQuantizationNoiseStandardDeviation(precisionBits)
```

```
quantizationNoiseStandardDeviation = 1.7206e-08
```

Compute Fixed-Point Types

In this example, assume that the designed system matrix A does not have full rank (there are fewer signals of interest than number of columns of matrix A), and the measured system matrix A has additive thermal noise that is larger than the quantization noise. The additive noise makes the measured matrix A have full rank.

Set $\sigma_{\text{noise}} = \sigma_{\text{thermal noise}}$.

```
noiseStandardDeviation = thermalNoiseStandardDeviation;
```

Use fixed.realQRMatrixSolveFixedpointTypes to compute fixed-point types.

```
T = fixed.realQRMatrixSolveFixedpointTypes(m,n,max_abs_A,max_abs_B,...
    precisionBits,noiseStandardDeviation)
```

```
T = struct with fields:
    A: [0x0 embedded.fi]
    B: [0x0 embedded.fi]
    X: [0x0 embedded.fi]
```

T.A is the type computed for transforming A to R in-place so that it does not overflow.

```
T.A
```

```
ans = 

[]

          DataTypeMode: Fixed-point: binary point scaling
```

```
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.B` is the type computed for transforming B to $Q'B$ in-place so that it does not overflow.

```
T.B
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 31
        FractionLength: 24
```

`T.X` is the type computed for the solution $X = A\backslash B$ so that there is a low probability that it overflows.

```
T.X
```

```
ans =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 36
        FractionLength: 24
```

Upper Bounds for R and C=Q'B

The upper bounds for R and $C = Q'B$ are computed using the following formulas, where m is the number of rows of matrices A and B.

$$\max(|R(:)|) \le \sqrt{m}\max(|A(:)|)$$

$$\max(|C(:)|) \le \sqrt{m}\max(|B(:)|)$$

These upper bounds are used to select a fixed-point type with the required number of bits of precision to avoid overflows.

```
upperBoundR = sqrt(m)*max_abs_A
```

```
upperBoundR = 17.3205
```

```
upperBoundQB = sqrt(m)*max_abs_B
```

```
upperBoundQB = 17.3205
```

Lower Bound for min(svd(A)) for Real A

A lower bound for min(svd(A)) is estimated by the `fixed.realSingularValueLowerBound` function using a probability that the estimate s is not greater than the actual smallest singular value. The default probability is 5 standard deviations below the mean. You can change this probability by specifying it as the last input parameter to the `fixed.realSingularValueLowerBound` function.

```
estimatedSingularValueLowerBound = fixed.realSingularValueLowerBound(m,n,noiseStandardDeviation)
```

```
estimatedSingularValueLowerBound = 0.0371
```

Simulate and Compare to the Computed Bounds

The bounds are within an order of magnitude of the simulated results. This is sufficient because the number of bits translates to a logarithmic scale relative to the range of values. Being within a factor of 10 is between 3 and 4 bits. This is a good starting point for specifying a fixed-point type. If you run the simulation for more samples, then it is more likely that the simulated results will be closer to the bound. This example uses a limited number of simulations so it doesn't take too long to run. For real-world system design, you should run additional simulations.

Define the number of samples, `numSamples`, over which to run the simulation.

```
numSamples = 1e4;
```

Run the simulation.

```
[actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A,max_abs_B,...
    numSamples,noiseStandardDeviation,T);
```

You can see that the upper bound on R compared to the measured simulation results of the maximum value of R over all runs is within an order of magnitude.

```
upperBoundR
```

```
upperBoundR = 17.3205
```

```
max(actualMaxR)
```

```
ans = 8.3029
```

You can see that the upper bound on $C = Q'B$ compared to the measured simulation results of the maximum value of $C = Q'B$ over all runs is also within an order of magnitude.

```
upperBoundQB
```

```
upperBoundQB = 17.3205
```

```
max(actualMaxQB)
```

```
ans = 2.5707
```

Finally, see that the estimated lower bound of min(svd(A)) compared to the measured simulation results of min(svd(A)) over all runs is also within an order of magnitude.

```
estimatedSingularValueLowerBound
```

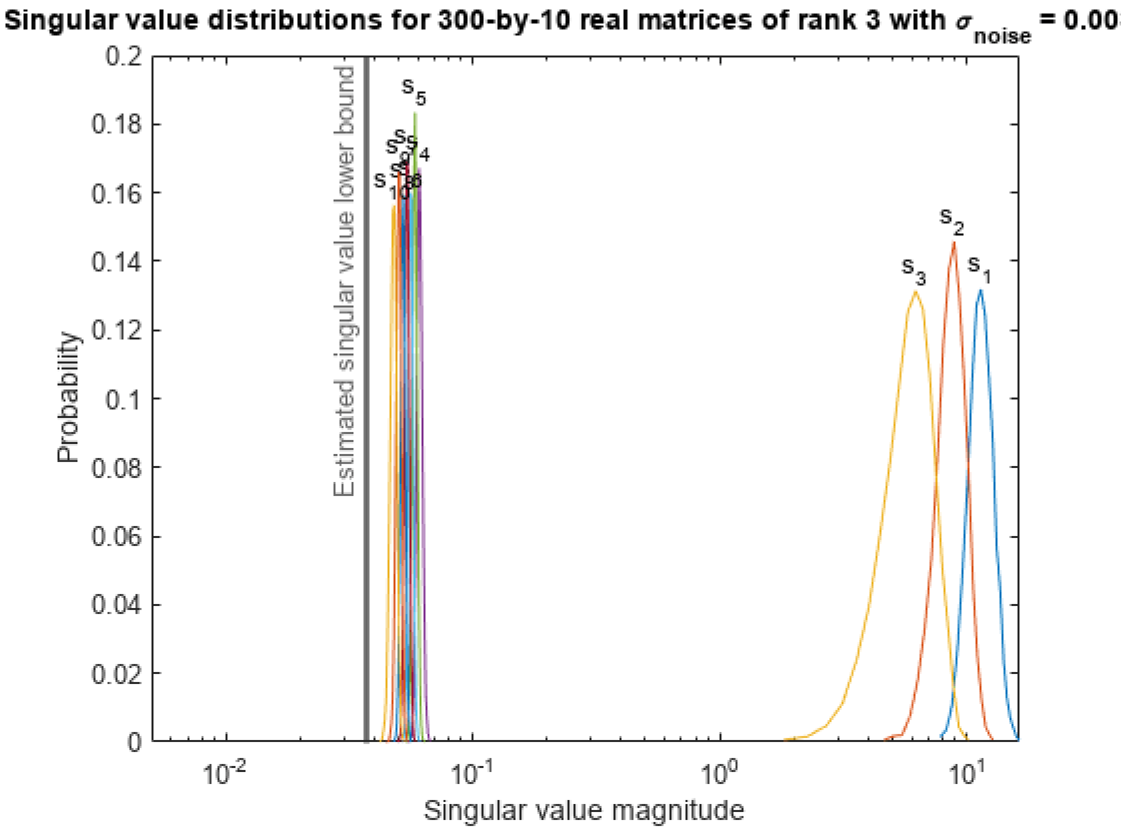
```
estimatedSingularValueLowerBound = 0.0371
```

```
actualSmallestSingularValue = min(singularValues,[],'all')
```

```
actualSmallestSingularValue = 0.0420
```

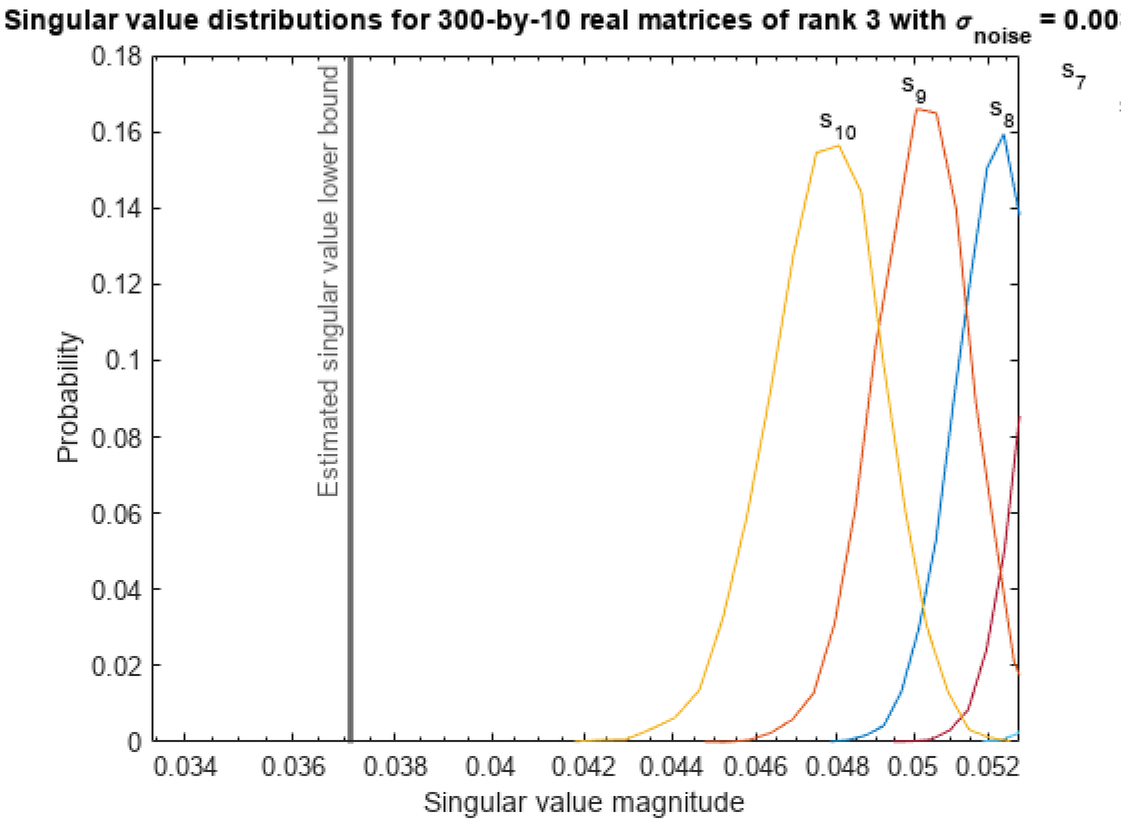
Plot the distribution of the singular values over all simulation runs. The distributions of the largest singular values correspond to the signals that determine the rank of the matrix. The distributions of the smallest singular values correspond to the noise. The derivation of the estimated bound of the smallest singular value makes use of the random nature of the noise.

```
clf
fixed.example.plot.singularValueDistribution(m,n,rankA,noiseStandardDeviation,...
    singularValues,estimatedSingularValueLowerBound,"real");
```



Zoom in to smallest singular value to see that the estimated bound is close to it.

```
xlim([estimatedSingularValueLowerBound*0.9, max(singularValues(n,:))]);
```



Estimate the largest value of the solution, X, and compare it to the largest value of X found during the simulation runs. The estimation is within an order of magnitude of the actual value, which is sufficient for estimating a fixed-point data type, because it is between 3 and 4 bits.

This example uses a limited number of simulation runs. With additional simulation runs, the actual largest value of X will approach the estimated largest value of X.

```
estimated_largest_X = fixed.realMatrixSolveUpperBoundX(m,n,max_abs_B,noiseStandardDeviation)
```

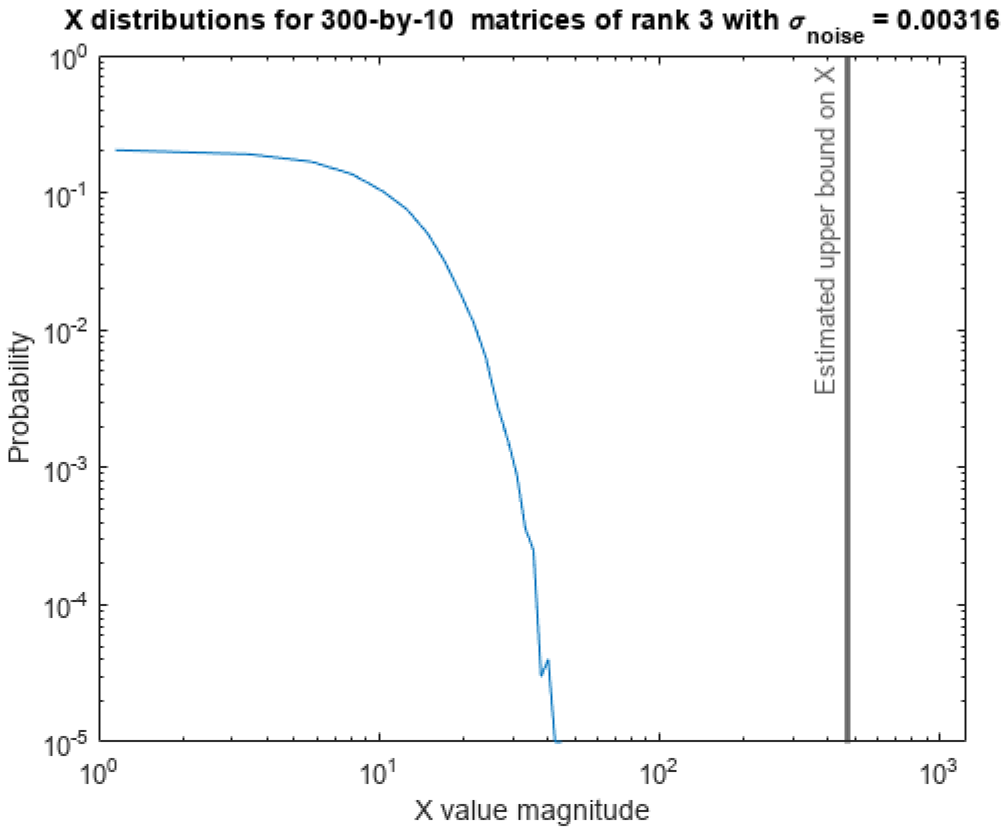
```
estimated_largest_X = 466.5772
```

```
actual_largest_X = max(abs(X_values),[],'all')
```

```
actual_largest_X = 44.8056
```

Plot the distribution of X values and compare it to the estimated upper bound for X.

```
clf
fixed.example.plot.xValueDistribution(m,n,rankA,noiseStandardDeviation,...
    X_values,estimated_largest_X,"real normally distributed random");
```



Supporting Functions

The `runSimulations` function creates a series of random matrices A and B of a given size and rank, quantizes them according to the computed types, computes the QR decomposition of A, and solves the equation $AX = B$. It returns the maximum values of $R = Q'A$ and $C = Q'B$, the singular values of A, and the values of X so their distributions can be plotted and compared to the bounds.

```
function [actualMaxR,actualMaxQB,singularValues,X_values] = runSimulations(m,n,p,rankA,max_abs_A
        numSamples,noiseStandardDeviation,T)
    precisionBits = T.A.FractionLength;
    A_WordLength = T.A.WordLength;
    B_WordLength = T.B.WordLength;
    actualMaxR = zeros(1,numSamples);
    actualMaxQB = zeros(1,numSamples);
    singularValues = zeros(n,numSamples);
    X_values = zeros(n,numSamples);
    for j = 1:numSamples
        A = max_abs_A*fixed.example.realRandomLowRankMatrix(m,n,rankA);
        % Adding normally distributed random noise makes A non-singular.
        A = A + fixed.example.realNormalRandomArray(0,noiseStandardDeviation,m,n);
        A = quantizenumeric(A,1,A_WordLength,precisionBits);
        B = fixed.example.realUniformRandomArray(-max_abs_B,max_abs_B,m,p);
        B = quantizenumeric(B,1,B_WordLength,precisionBits);
        [Q,R] = qr(A,0);
        C = Q'*B;
        X = R\C;
        actualMaxR(j) = max(abs(R(:)));
```

```
        actualMaxQB(j) = max(abs(C(:)));
        singularValues(:,j) = svd(A);
        X_values(:,j) = X;
    end
end
```

References

1 Thomas A. Bryan and Jenna L. Warren. “Systems and Methods for Design Parameter Selection”. Patent pending. U.S. Patent Application No. 16/947,130. 2020.

2 Perform QR Factorization Using CORDIC. Derivation of the bound on growth when computing QR. MathWorks. 2010.

3 Zizhong Chen and Jack J. Dongarra. “Condition Numbers of Gaussian Random Matrices”. In: SIAM J. Matrix Anal. Appl. 27.3 (July 2005), pp. 603–620. issn: 0895-4798. doi: 10.1137/040616413. url: https://dx.doi.org/10.1137/040616413.

4 Bernard Widrow. “A Study of Rough Amplitude Quantization by Means of Nyquist Sampling Theory”. In: IRE Transactions on Circuit Theory 3.4 (Dec. 1956), pp. 266–276.

5 Bernard Widrow and István Kollár. Quantization Noise – Roundoff Error in Digital Computation, Signal Processing, Control, and Communications. Cambridge, UK: Cambridge University Press, 2008.

6 Gene H. Golub and Charles F. Van Loan. Matrix Computations. Second edition. Baltimore: Johns Hopkins University Press, 1989.

Suppress `mlint` warnings in this file.

```
%#ok<*NASGU>
%#ok<*ASGLU>
```

Input Arguments

`m` — Number of rows in matrix

positive integer-valued scalar

Number of rows in matrix, specified as a positive integer-valued scalar. The number of rows, `m`, must be greater than or equal to the number of columns, `n`.

Data Types: `double`

`n` — Number of columns in matrix

positive integer-valued scalar

Number of columns in matrix, specified as a positive integer-valued scalar. The number of rows, `m`, must be greater than or equal to the number of columns, `n`.

Data Types: `double`

`noiseStandardDeviation` — Standard deviation of additive random noise in matrix

scalar

Standard deviation of additive random noise in matrix, specified as a scalar.

Data Types: `double`

p_s_n — Probability that estimate of lower bound is larger than actual smallest singular value of matrix
2.8665e-07 (default) | scalar

Probability that estimate of lower bound is larger than actual smallest singular value of matrix, specified as a scalar.

If `p_s_n` is not supplied or empty, then the default of `p_s_n = (1/2)*(1+erf(-5/sqrt(2))) = 2.8665e-07` is used, which is 5 standard deviations below the mean, so the probability that the estimated lower bound for the smallest singular value is less than the actual smallest singular value is `1 - p_s = 0.99999971 - p_s = 0.9999997`.

Data Types: `double`

regularizationParameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar. Small, positive values of the regularization parameter can improve the conditioning of the problem and reduce the variance of the estimates. While biased, the reduced variance of the estimate often results in a smaller mean squared error when compared to least-squares estimates.

`regularizationParameter` is the Tikhonov regularization parameter of the matrix $\begin{bmatrix} \lambda I_n \\ A \end{bmatrix}$ where λ is the `regularizationParameter`, A is an m-by-n matrix with m >= n, and I = `eye(n)`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

s_n — Estimate of lower bound for smallest singular value of real-valued matrix
scalar

Estimate of lower bound for smallest singular value of real-valued matrix, returned as a scalar.

Tips

- Use `fixed.realSingularValueLowerBound` to estimate the smallest singular value of a matrix to estimate a bound for max(|X(:)|). For example, in `fixed.realQRMatrixSolveFixedpointTypes`, the elements of X=R\(Q'B) are bounded in magnitude by

  $$\max(|X(:)|) \le \frac{\sqrt{m}\max(|B(:)|)}{\min(\mathrm{svd}(A))} \le \frac{\sqrt{m}\max(|B(:)|)}{s}$$

  with probability 1-p_s.
- max(|X(:)|) is smaller when the denominator in the above equation is larger.
- If nothing else is known about a matrix, then generally, the smallest singular value will be larger if:

  - there is additive random noise.

- the number of rows, m, is much larger than the number of columns, n.
- If the noise standard deviation is not known, you can approximate it as the standard deviation of the quantization error. You can compute the quantization error using `fixed.realQuantizationNoiseStandardDeviation`.
- For s to be a useful bound on the smallest singular value of A, the probability that s is greater than the smallest singular value of A should be small. A practical value to use is

  $$p_s = (1/2) \cdot (1 + \mathrm{erf}(-5/\sqrt{2})) \approx 3 \cdot 10^{-7}$$

  which is 5 standard deviations below the mean, so the probability that the estimated bound for the smallest singular value is less than the actual smallest singular value is 1-p_s ≈ 0.9999997.
- `fixed.realSingularValueLowerBound` is used in these functions.
  - `fixed.realQlessQRMatrixSolveFixedpointTypes`
  - `fixed.realQRMatrixSolveFixedpointTypes`

Algorithms

Given a m-by-n real-valued matrix A and standard deviation σ_N of additive random noise on the elements of A, you can compute an estimate of a lower bound for the smallest singular value of A, s, such that the probability, p_s, of s being greater than the smallest singular value of A using this formula [1][2].

$$s = \sigma_N \sqrt{2\gamma^{-1}\left(\frac{p_s \Gamma(m-n+1)\Gamma(n/2)}{2^{m-n}\Gamma\left(\frac{m+1}{2}\right)\Gamma\left(\frac{m-n+1}{2}\right)}, \frac{m-n+1}{2}\right)}$$

Version History

Introduced in R2021b

R2022a: Support for Tikhonov regularization parameter

The `fixed.realSingularValueLowerBound` function now supports the Tikhonov regularization parameter, “regularizationParameter” on page 4-0 .

References

[1] Bryan, Thomas A. and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[2] Chen, Zizhong and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no. 3 (July 2005): 603-620. https://doi.org/10.1137/040616413.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also
fixed.realQRMatrixSolveFixedpointTypes | fixed.realQuantizationNoiseStandardDeviation | fixed.realQlessQRMatrixSolveFixedpointTypes | fixed.realQRMatrixSolveFixedpointTypes

fixed.singularValueUpperBound

Upper bound of largest singular value of matrix

Syntax

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max_abs_A)
svdUpperBound = fixed.singularValueUpperBound(m,n,max_abs_A,
regularizationParameter)
```

Description

`svdUpperBound = fixed.singularValueUpperBound(m,n,max_abs_A)` returns an upper bound of the largest singular value of an `m`-by-`n` matrix `A`, where `m >= n` and `max_abs_A >= max(abs(A(:)))`.

`svdUpperBound = fixed.singularValueUpperBound(m,n,max_abs_A, regularizationParameter)` returns an upper bound of the largest singular value of the matrix `[regularizationParameter*eye(n); A]`, where `A` is an `m`-by-`n` matrix with `m >= n`.

Examples

Upper Bound of Largest Singular Value of Real-Valued Matrix

Define a real-valued matrix, `A`.

```
m = 5;
n = 3;
A = ones(m,n);
max_abs_A = 1;
```

Determine an upper bound for the largest singular value of the matrix.

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max_abs_A)
```

```
svdUpperBound = 3.8730
```

Compare to the actual largest singular value of the matrix.

```
actual_largest_singular_value = max(svd(A))
```

```
actual_largest_singular_value = 3.8730
```

Upper Bound of Largest Singular Value of a Low Rank Matrix with Regularization

Use the helper function `realRandomLowRankMatrix` to define a real-valued, low rank matrix `A`.

```
m = 300;
n = 10;
```

```
rankA = 3;
A = realRandomLowRankMatrix(m,n,rankA);
```

Determine an upper bound for the largest singular value of the matrix of the Tikhonov regularized problem.

```
regularizationParameter = 0.01;
A = [regularizationParameter*eye(n);A];
svdUpperBound = fixed.singularValueUpperBound(m,n,max(abs(A(:))),regularizationParameter)
```

```
svdUpperBound = 54.7823
```

Compare to the actual largest singular value of the matrix.

```
actual_largest_singular_value = max(svd(A))
```

```
actual_largest_singular_value = 13.3424
```

Input Arguments

m — Number of rows in matrix A
positive integer-valued scalar

Number of rows in matrix A, specified as a positive integer-valued scalar. The number of rows, m, must be greater than or equal to the number of columns, n.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

n — Number of columns in matrix A
positive integer-valued scalar

Number of columns in matrix A, specified as a positive integer-valued scalar. The number of rows, m, must be greater than or equal to the number of columns, n.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

max_abs_A — Maximum of absolute value of matrix A
scalar

Maximum of absolute value of matrix A, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

regularizationParameter — Regularization parameter
0 (default) | nonnegative scalar

Regularization parameter, specified as a nonnegative scalar.

Data Types: single | double

Algorithms

The upper bound for the largest singular value of matrix A is `svdUpperBound = sqrt(m*n)*max(abs(A(:)))`. If there is a regularization parameter, then the upper bound is `svdUpperBound = sqrt(m*n)*max(abs(A(:))) + abs(regularizationParameter)` [1][2][3].

Version History

Introduced in R2022b

References

[1] Bryan, Thomas A., Jenna L. Warren, Brenda Zhuang, and Jessica Clayton. Continuation in Part for "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2022.

[2] Bryan, Thomas A., and Jenna L. Warren. "Systems and Methods for Design Parameter Selection." U.S. Patent Application No. 16/947, 130. 2020.

[3] Chen, Zizhong, and Jack J. Dongarra. "Condition Numbers of Gaussian Random Matrices." *SIAM Journal on Matrix Analysis and Applications* 27, no. 3 (July 2005): 603—620. https://doi.org/10.1137/040616413.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`fixed.realSingularValueLowerBound` | `fixed.complexSingularValueLowerBound` | `fixed.svd`

fixed.svd

Fixed-point Golub-Kahan-Reinsch singular value decomposition

Syntax

```
S = fixed.svd(A)
[U,S,V] = fixed.svd(A)
[ ___ ] = fixed.svd(A,"econ")
[ ___ ] = fixed.svd(A,0)
[ ___ ] = fixed.svd( ___ ,sigmaForm)
```

Description

`S = fixed.svd(A)` returns the singular values of matrix `A` in descending order.

`[U,S,V] = fixed.svd(A)` performs a singular value decomposition of matrix `A` such that `A = U*S*V'`. `S` is a diagonal matrix of the same dimension as `A` with nonnegative diagonal elements in decreasing order. `U` and `V` are unitary matrices.

`[ ___ ] = fixed.svd(A,"econ")` produces an economy-size decomposition of `A`. If `A` is an *m*-by-*n* matrix, then:

- *m* > *n* — Only the first *n* columns of `U` are computed and `S` is *n*-by-*n*.
- *m* = *n* — `fixed.svd(A,"econ")` is equivalent to `fixed.svd(A)`.
- *m* < *n* — Only the first *m* columns of `V` are computed, and `S` is *m*-by-*m*.

`[ ___ ] = fixed.svd(A,0)` produces a different economy-size decomposition of `A`. If `A` is an *m*-by-*n* matrix, then:

- *m* > *n* — `fixed.svd(A,0)` is equivalent to `fixed.svd(A,"econ")`.
- *m* <= *n* — `fixed.svd(A,0)` is equivalent to `fixed.svd(A)`.

Syntax is not recommended. Use the `"econ"` option instead.

`[ ___ ] = fixed.svd( ___ ,sigmaForm)` optionally specifies the output format for the singular values. You can use this option with any of the previous input or output combinations. Specify `"vector"` to return the singular values as a column vector. Specify `"matrix"` to return the singular values in a diagonal matrix.

Examples

Singular Values of Fixed-Point Matrix

Compute the singular values of a full rank scaled-double matrix.

```
A = [1 0 1; -1 -2 0; 0 1 -1];
```

Define fixed-point types that will never overflow. First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Then, define the integer length based on the value of the upper bound, with one additional bit for the sign and another additional bit for intermediate CORDIC growth. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(3,3,max(abs(A(:))));
integerLength = ceil(log2(svdUpperBound)) + 2;
wordLength = 16;
fractionLength = wordLength - integerLength;
```

Cast the matrix A to the scaled-double type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','ScaledDouble');
A = cast(A,'like',T.A)
```

```
A =
     1     0     1
    -1    -2     0
     0     1    -1


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Compute the singular values.

```
s = fixed.svd(A)
```

```
s =
    2.4605
    1.6996
    0.2391


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

The singular values are returned in a column vector in decreasing order, and have the same data type as A.

Fixed-Point Singular Value Decomposition

Find the singular value decomposition of the rectangular fixed-point matrix A.

Define the rectangular matrix A.

```
m = 4;
n = 2;
rng('default');
A = 10*randn(m,n);
```

Define fixed-point types that will never overflow. First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Then, define the integer length based on the value of the upper bound, with one additional bit for the sign and another additional bit for intermediate CORDIC growth. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max(abs(A(:))));
integerLength = ceil(log2(svdUpperBound)) + 2;
wordLength = 32;
fractionLength = wordLength - integerLength;
```

Cast the matrix A to the signed fixed-point type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','Fixed');
A = cast(A,'like',T.A)
```

```
A =
    5.3767    3.1877
   18.3389  -13.0769
  -22.5885   -4.3359
    8.6217    3.4262

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24
```

Find the singular value decomposition of the fixed-point matrix A.

```
[U,S,V] = fixed.svd(A)
```

```
U =
    0.1590    0.2717   -0.9387   -0.1403
    0.6397   -0.7548   -0.1219    0.0790
   -0.7049   -0.5057   -0.3224    0.3787
    0.2619    0.3174         0    0.9114

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S =
   31.0141         0
         0   14.1289
         0         0
         0         0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 24

V =
    0.9920    0.1259
   -0.1259    0.9920

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Confirm the relation A = U*S*V'.

```
U*S*V'
```

```
ans = 
    5.3767    3.1877
   18.3389  -13.0769
  -22.5885   -4.3359
    8.6217    3.4262

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 99
        FractionLength: 84
```

Economy-Size Decomposition

Calculate the complete and economy-size decomposition of a rectangular fixed-point matrix.

Define the matrix A.

```
A = [1 2; 3 4; 5 6; 7 8];
```

Define fixed-point types that will never overflow. First, use the fixed.singularValueUpperBound function to determine the upper bound on the singular values. Then, define the integer length based on the value of the upper bound, with one additional bit for the sign and another additional bit for intermediate CORDIC growth. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(4,2,max(abs(A(:))));
integerLength = ceil(log2(svdUpperBound)) + 2;
wordLength = 32;
fractionLength = wordLength - integerLength;
```

Cast the matrix A to the signed fixed-point type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','Fixed');
A = cast(A,'like',T.A)
```

```
A = 
     1     2
     3     4
     5     6
     7     8

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 25
```

Compute the complete decomposition.

```
[U,S,V] = fixed.svd(A)
```

```
U = 
   -0.1525    0.8226   -0.4082    0.3651
```

```
   -0.3499    0.4214    0.8165   -0.1826
   -0.5474    0.0201   -0.4082   -0.7303
   -0.7448   -0.3812         0    0.5477

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S = 
   14.2691         0
         0    0.6268
         0         0
         0         0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 25

V = 
   -0.6414   -0.7672
   -0.7672    0.6414

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Compute the economy-size decomposition.

```
[U,S,V] = fixed.svd(A,"econ")
```

```
U = 
   -0.1525    0.8226
   -0.3499    0.4214
   -0.5474    0.0201
   -0.7448   -0.3812

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S = 
   14.2691         0
         0    0.6268

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 25

V = 
   -0.6414   -0.7672
   -0.7672    0.6414

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
            WordLength: 32
        FractionLength: 30
```

Since A is 4-by-2, `fixed.svd(A,"econ")` returns fewer columns in U and fewer rows in S compared to a complete decomposition. Extra rows of zeros in S are excluded, along with the corresponding columns in U that would multiply with those zeros in the expression `A = U*S*V'`.

Control Singular Value Output Format

Create a 3-by-3 magic square matrix and calculate the singular value decomposition. By default, the `fixed.svd` function returns the singular values in a diagonal matrix when you specify multiple outputs.

Define the matrix A.

```
m = 3;
n = m;
A = magic(m);
```

Define fixed-point types that will never overflow. First, use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Then, define the integer length based on the value of the upper bound, with one additional bit for the sign and another additional bit for intermediate CORDIC growth. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max(abs(A(:))));
integerLength = ceil(log2(svdUpperBound)) + 2;
wordLength = 32;
fractionLength = wordLength - integerLength;
```

Cast the matrix A to the signed fixed-point type.

```
T.A = fi([],1,wordLength,fractionLength,'DataType','Fixed');
A = cast(A,'like',T.A)
```

```
A = 
     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 25
```

Compute the singular value decomposition.

```
[U,S,V] = fixed.svd(A)
```

```
U = 
    0.5774   -0.7071   -0.4082
    0.5774    0.0000    0.8165
    0.5774    0.7071   -0.4082

          DataTypeMode: Fixed-point: binary point scaling
```

```
           Signedness: Signed
           WordLength: 32
       FractionLength: 30

S = 
   15.0000         0         0
         0    6.9282         0
         0         0    3.4641

         DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 32
       FractionLength: 25

V = 
    0.5774   -0.4082   -0.7071
    0.5774    0.8165   -0.0000
    0.5774   -0.4082    0.7071

         DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 32
       FractionLength: 30
```

Specify the `"vector"` option to return the singular values in a column vector.

```
[U,S,V] = fixed.svd(A,"vector")
```

```
U = 
    0.5774   -0.7071   -0.4082
    0.5774    0.0000    0.8165
    0.5774    0.7071   -0.4082

         DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 32
       FractionLength: 30

S = 
   15.0000
    6.9282
    3.4641

         DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 32
       FractionLength: 25

V = 
    0.5774   -0.4082   -0.7071
    0.5774    0.8165   -0.0000
    0.5774   -0.4082    0.7071

         DataTypeMode: Fixed-point: binary point scaling
           Signedness: Signed
           WordLength: 32
       FractionLength: 30
```

If you specify one output argument, such as `S = fixed.svd(A)`, then `fixed.svd` switches behavior to return the singular values in a column vector by default. In that case, you can specify the `"matrix"` option to return the singular values as a diagonal matrix.

Compute Fixed-Point Singular Value Decomposition and Generate Code

Compute the fixed-point singular value decomposition, verify the results, and generate purely integer C code.

Define the input matrix `A`.

```
m = 10;
n = 4;
rng('default');
A = 10*randn(m,n);
```

The `fixed.svd` function also accepts complex inputs.

```
A = 10*complex(rand(m,n),rand(m,n));
```

Define fixed-point types that will never overflow. Use the `fixed.singularValueUpperBound` function to determine the upper bound on the singular values. Define the integer length based on the value of the upper bound, with one additional bit for the sign and another additional bit for intermediate CORDIC growth. Compute the fraction length based on the integer length and the desired word length.

```
svdUpperBound = fixed.singularValueUpperBound(m,n,max(abs(A(:))));
integerLength = ceil(log2(svdUpperBound)) + 2;
wordLength = 32;
fractionLength = wordLength - integerLength;
```

Specify the desired data type for the input matrix `A`.

```
dataType = 'Fixed';
T.A = fi([],1,wordLength,fractionLength,'DataType',dataType);
disp(T.A)
```

```
[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 23
```

Cast the matrix `A` to the signed fixed-point type.

```
A = cast(A,'like',T.A);
```

Generate a MATLAB executable (MEX) file for execution speed. Use the `"econ"` flag to compute the economy-size singular-value decomposition. Use the `"vector"` flag to return the singular values as a vector, `s`. The flags must be constant for code generation. Use the `-nargout 3` flag to indicate to the `codegen` function that it is to generate code for the three-output syntax.

```
codegen +fixed/svd -o fixedSVD_mex -args {A,coder.Constant("econ"),coder.Constant("vector")} -na
```

```
Code generation successful.
```

Run the MEX file.

```
[U,s,V] = fixedSVD_mex(A,"econ","vector")
```

```
U = 
   0.2509 + 0.1236i   0.1980 + 0.1578i   0.1745 + 0.0268i   0.0755 - 0.2443i
   0.1601 + 0.2073i  -0.3227 - 0.1684i   0.1420 - 0.3385i   0.3686 - 0.1899i
   0.2937 + 0.1868i   0.0574 - 0.2108i   0.0884 - 0.0633i  -0.1079 + 0.0866i
   0.2071 + 0.3019i   0.0678 + 0.1740i  -0.3790 + 0.1518i  -0.3808 + 0.1483i
   0.2262 + 0.2405i   0.5884 - 0.1889i  -0.0693 - 0.3624i   0.0547 - 0.1581i
   0.2435 + 0.2111i  -0.0568 + 0.3536i  -0.1593 + 0.1445i   0.3714 - 0.0026i
   0.2195 + 0.2411i  -0.0713 - 0.2517i  -0.2848 + 0.2641i   0.4055 - 0.0481i
   0.1461 + 0.3184i  -0.1657 + 0.0477i   0.0478 + 0.2812i  -0.2972 - 0.0396i
   0.2204 + 0.1354i  -0.2868 - 0.0745i   0.3628 - 0.2391i  -0.0565 + 0.1251i
   0.1863 + 0.2334i  -0.1495 + 0.0491i   0.1422 - 0.1789i  -0.3594 + 0.0775i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

s = 
   45.9444
   16.1174
   10.7897
    8.3153

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 23

V = 
   0.5637 + 0.0000i  -0.4038 + 0.0000i  -0.2061 + 0.0000i  -0.6904 + 0.0000i
   0.4261 + 0.0228i  -0.3782 - 0.4565i   0.0076 + 0.3326i   0.5667 + 0.1863i
   0.4980 + 0.0276i   0.3223 + 0.4193i  -0.5081 - 0.2300i   0.3698 - 0.1540i
   0.5014 + 0.0041i   0.4531 - 0.0243i   0.7280 - 0.0758i  -0.0729 + 0.0402i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Verify the singular values. Since singular values are unique, you can use the svd function to verify that `fixed.svd` gives a comparable result within the precision of the selected fixed-point type.

```
sExpected = svd(double(A))
```

```
sExpected = 4×1

   45.9444
   16.1174
   10.7897
    8.3153
```

```
singularValueRelativeError = norm(double(s)-double(sExpected))/norm(double(sExpected))
```

```
singularValueRelativeError = 3.7197e-07
```

Singular vectors are not unique. You can verify the singular vectors by confirming that `A ≈ U*S*V'` and that the singular vector matrices are orthonormal.

First, expand the singular value vector `s` into matrix `S`.

```
S = zeros(size(U,2),size(V,2),'like',s);
for i = 1:min(m,n)
    S(i,i) = s(i);
end
S
```

```
S =
   45.9444         0         0         0
         0   16.1174         0         0
         0         0   10.7897         0
         0         0         0    8.3153

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 23
```

Verify that `U*S*V'` is approximately equal to `A`.

```
decompositionRelativeError = norm(double(U*S*V')-double(A))/norm(double(A))
```

```
decompositionRelativeError = 3.3811e-07
```

`U` and `V` are orthonormal. Verify that `U'U` and `V'V` are approximately equal to the identity matrix.

```
UtransposeU = double(U'*U)
```

```
UtransposeU = 4×4 complex

   1.0000 + 0.0000i  -0.0000 + 0.0000i  -0.0000 + 0.0000i   0.0000 + 0.0000i
  -0.0000 - 0.0000i   1.0000 + 0.0000i  -0.0000 + 0.0000i   0.0000 + 0.0000i
  -0.0000 - 0.0000i  -0.0000 - 0.0000i   1.0000 + 0.0000i  -0.0000 - 0.0000i
   0.0000 - 0.0000i   0.0000 - 0.0000i  -0.0000 + 0.0000i   1.0000 + 0.0000i
```

```
VtransposeV = double(V'*V)
```

```
VtransposeV = 4×4 complex

   1.0000 + 0.0000i  -0.0000 - 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
  -0.0000 + 0.0000i   1.0000 + 0.0000i  -0.0000 + 0.0000i  -0.0000 + 0.0000i
   0.0000 - 0.0000i  -0.0000 - 0.0000i   1.0000 + 0.0000i   0.0000 - 0.0000i
   0.0000 - 0.0000i  -0.0000 - 0.0000i   0.0000 + 0.0000i   1.0000 + 0.0000i
```

Generate C code. If the input is fixed point, you can verify that the generated C code consists only of integer types.

```
cfg = coder.config('lib');

if isfi(A) && isfixed(A)
    cfg.PurelyIntegerCode = true;
```

```
end
codegen +fixed/svd -args {A, coder.Constant("econ"), coder.Constant("vector")} -config cfg -narg
```

```
Warning: C Compiler produced warnings. See the build log for further details.

Code generation successful (with warnings): To view the report, open('codegen\lib\svd\html\repor
```

The MATLAB code for `fixed.svd` does not appear in the code generation report because `fixed.svd` is a MATLAB toolbox function.

Input Arguments

A — Input matrix
matrix

Input matrix, specified as a matrix. `A` can be a signed fixed-point `fi`, a signed scaled double `fi`, `double`, or `single` data type.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`sigmaForm` — Output format of singular values
`"vector"` | `"matrix"`

Output format of singular values, specified as one of these values:

- `"vector"` — `S` is a column vector. This behavior is the default when you specify one output, `S = fixed.svd(A)`.
- `"matrix"` — `S` is a diagonal matrix. This behavior is the default when you specify multiple outputs, `[U,S,V] = fixed.svd(A)`.

Example: `[U,S,V] = fixed.svd(X,"vector")` returns `S` as a column vector instead of a diagonal matrix.

Example: `S = fixed.svd(X,"matrix")` returns `S` as a diagonal matrix instead of a column vector.

Data Types: `char` | `string`

Output Arguments

U — Left singular vectors
matrix

Left singular vectors, returned as the columns of a matrix.

For fixed-point and scaled-double inputs, `U` is returned as a signed fixed-point or scaled-double `fi` with the same word length as `A` and fraction length equal to two less than the word length. One of these integer bits is used for the sign. The other integer bit allows `+1` to be represented exactly.

For floating-point input, `U` has the same data type as `A`.

S — Singular values
diagonal matrix | column vector

Singular values, returned as a diagonal matrix or column vector. The singular values are nonnegative and returned in decreasing order. The singular values `S` have the same data type as `A`.

`V` — Right singular vectors
matrix

Right singular vectors, returned as the columns of a matrix.

For fixed-point input and scaled-double inputs, `V` is returned as a signed fixed-point or scaled-double `fi` with the same word length as `A` and fraction length equal to two less than the word length. One of these integer bits is used for the sign. The other integer bit allows `+1` to be represented exactly.

For floating-point input, `V` has the same data type as `A`. One of these integer bits is used for the sign, and the other integer bit allows `+1` to be represented exactly.

Tips

The `fixed.svd` function allows full control over the fixed-point types. `fixed.svd` computes in-place in the same data type as the input, which may overflow but will produce efficient code. The `svd` function adjusts the data type of a fixed-point input to avoid overflow and increase precision.

Algorithms

The Golub-Kahan-Reinsch algorithm is a sequential method that performs well on serial computers. For parallel computing, as in FPGA and ASIC applications, use the `fixed.jacobiSVD` function.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

`fixed.svd` generates efficient, purely integer C code.

See Also

`svd` | `svd`

Topics

"Singular Values"

fixpt_instrument_purge

Remove corrupt fixed-point instrumentation from model

Compatibility

Note `fixpt_instrument_purge` will be removed in a future release.

Syntax

```
fixpt_instrument_purge
fixpt_instrument_purge(modelName, interactive)
```

Description

The `fixpt_instrument_purge` script finds and removes fixed-point instrumentation from a model left by the Fixed-Point Tool and the fixed-point autoscaling script. The Fixed-Point Tool and the fixed-point autoscaling script each add callbacks to a model. For example, the Fixed-Point Tool appends commands to model-level callbacks. These callbacks make the Fixed-Point Tool respond to simulation events. Similarly, the autoscaling script adds instrumentation to some parameter values that gathers information required by the script.

Normally, these types of instrumentation are automatically removed from a model. The Fixed-Point Tool removes its instrumentation when the model is closed. The autoscaling script removes its instrumentation shortly after it is added. However, there are cases where abnormal termination of a model leaves fixed-point instrumentation behind. The purpose of `fixpt_instrument_purge` is to find and remove fixed-point instrumentation left over from abnormal termination.

`fixpt_instrument_purge(modelName, interactive)` removes instrumentation from model `modelName`. `interactive` is `true` by default, which prompts you to make each change. When `interactive` is set to `false`, all found instrumentation is automatically removed from the model.

Version History

Introduced before R2006a

See Also

`autofixexp` | `fxptdlg`

floor

Round toward negative infinity

Syntax

```
y = floor(a)
```

Description

`y = floor(a)` rounds `fi` object `a` to the nearest integer in the direction of negative infinity and returns the result in `fi` object `y`.

Examples

Use `floor` on a Signed `fi` Object

The following example demonstrates how the `floor` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi,1,8,3)
```

```
a = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

```
y = floor(a)
```

```
y = 
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 5
        FractionLength: 0
```

The following example demonstrates how the `floor` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a = 
    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
y = floor(a)
```

```
y =
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Compare Rounding Methods

The functions `ceil`, `fix`, and `floor` differ in the way they round `fi` objects:

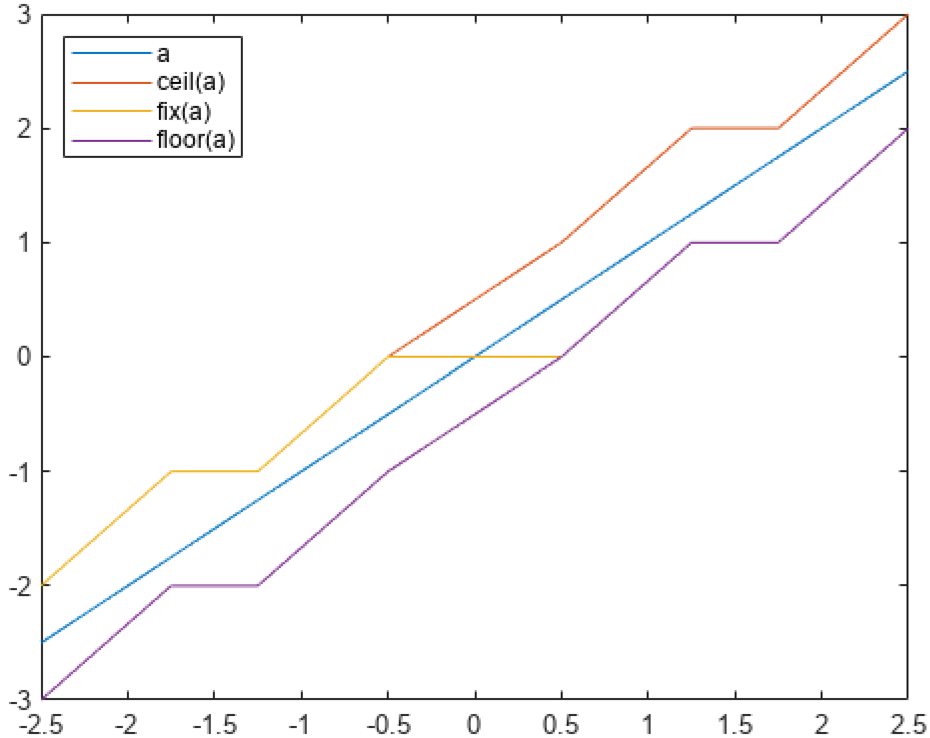
- The `ceil` function rounds values to the nearest integer toward positive infinity.
- The `fix` function rounds values to the nearest integer toward zero.
- The `floor` function rounds values to the nearest integer toward negative infinity.

This example illustrates these differences for a given `fi` input object `a`.

```
a = fi([-2.5,-1.75,-1.25,-0.5,0.5,1.25,1.75,2.5]');
y = [a ceil(a) fix(a) floor(a)]
```

```
y =
   -2.5000   -2.0000   -2.0000   -3.0000
   -1.7500   -1.0000   -1.0000   -2.0000
   -1.2500   -1.0000   -1.0000   -2.0000
   -0.5000         0         0   -1.0000
    0.5000    1.0000         0         0
    1.2500    2.0000    1.0000    1.0000
    1.7500    2.0000    1.0000    1.0000
    2.5000    3.0000    2.0000    2.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
plot(a,y); legend('a','ceil(a)','fix(a)','floor(a)','location','NW');
```



Input Arguments

a — Input `fi` array
scalar | vector | matrix | multidimensional array

Input `fi` array, specified as scalar, vector, matrix, or multidimensional array.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`floor` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Data Types: `fi`
Complex Number Support: Yes

Algorithms

- `y` and `a` have the same `fimath` object and `DataType` property.
- When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.

- When the fraction length of a is positive, the fraction length of y is 0, its sign is the same as that of a, and its word length is the difference between the word length and the fraction length of a, plus one bit. If a is signed, then the minimum word length of y is 2. If a is unsigned, then the minimum word length of y is 1.

Version History

Introduced in R2008a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

ceil | convergent | fix | nearest | round

floorDiv

Round the result of division toward negative infinity

Syntax

```
y = floorDiv(x,d)
y = floorDiv(x,d,m)
```

Description

`y = floorDiv(x,d)` returns the result of `x/d` rounded to the nearest integer value in the direction of negative infinity.

`y = floorDiv(x,d,m)` returns the result of `x/d` rounded to the nearest multiple of `m` in the direction of negative infinity.

The datatype of `y` is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of `x`, and the values of `d` and `m`.

Examples

Divide and Round to Floor

Perform a division operation and round to the nearest integer value in the direction of negative infinity.

```
floorDiv(int16(201),10)
```

```
ans = 
    20

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Perform a division operation and round to the nearest multiple of 7 in the direction of negative infinity.

```
floorDiv(int16(201),10,7)
```

```
ans = 
    14

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Divide and Generate Code

Define a function that uses floorDiv.

```
function y = floorDiv_example(x,d)
y = floorDiv(x,d);
end
```

Define inputs and execute the function in MATLAB®.

```
x = fi(pi);
d = fi(2);
y = floorDiv_example(x,d)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

To generate code for this function, the denominator d must be defined as a constant.

```
codegen floorDiv_example -args {x, coder.Constant(d)}
```

```
Code generation successful.
```

Alternatively, you can define the denominator, d, as constant in the body of the code.

```
function y = floorDiv10(x)
y = floorDiv(x,10);
end
```

```
x = fi(5*pi);
y = floorDiv10(x)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

```
codegen floorDiv10 -args {x}
```

```
Code generation successful.
```

Input Arguments

x — Dividend
scalar

Dividend, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

d — Divisor
scalar

Divisor, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

m — Value to round to nearest multiple of
1 (default) | scalar

Value to round to nearest multiple of, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

Output Arguments

y — Result of division and round to floor
scalar

Result of division and round to floor, returned as a scalar.

The datatype of y is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of x, and the values of d and m.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

To generate code, the denominator d must be declared as constant.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also
ceilDiv | fixDiv | nearestDiv

fma

Multiply and add using fused multiply add approach

Syntax

```
X = fma(A, B, C)
```

Description

`X = fma(A, B, C)` computes $A.*B+C$ using a fused multiply add approach. Fused multiply add operations round only once, often making the result more accurate than performing a multiplication operation followed by an addition.

Examples

Multiply and Add Three Inputs Using Fused Multiply Add

This example shows how to use the `fma` function to calculate $A \times B + C$ using a fused multiply add approach.

Define the inputs and use the `fma` function to compute the multiply add operation.

```
a = half(10);
b = half(10);
c = half(2);
x = fma(a, b, c)
```

```
x =

  half

   102
```

Compare the result of the `fma` function with the two-step approach of computing the product and then the sum.

```
temp = a * b;
x = temp + c
```

```
x =

  half

   102
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a floating-point scalar, vector, matrix, or multidimensional array. When A and B are matrices, fma performs element-wise multiplication followed by addition.

Data Types: single | double | half

B — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a floating-point scalar, vector, matrix, or multidimensional array. When A and B are matrices, fma performs element-wise multiplication followed by addition.

Data Types: single | double | half

C — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a floating-point scalar, vector, matrix, or multidimensional array.

Data Types: single | double | half

Output Arguments

X — Result of multiply and add operation
scalar | vector | matrix | multidimensional array

Result of multiply and add operation, *A.*B+C*, returned as a scalar, vector, matrix, or multidimensional array.

Version History

Introduced in R2019a

See Also

half

for

`for` loop to repeat specified number of times

Syntax

```
for index = values
   statements
end
```

Description

`for index = values, statements, end` executes a group of statements in a loop for a specified number of times.

If a `colon, :` operation with `fi` objects is used as the index, then the `fi` objects must be whole numbers.

Refer to the MATLAB `for` reference page for more information.

Examples

Use `fi` in a For Loop

Use a `fi` object as the index of a for loop.

```
a = fi(1,0,8,0);
b = fi(2,0,8,0);
c = fi(10,0,8,0);

for x = a:b:c
 x
end
```

```
x =
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0

x =
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0

x =
     5
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0

x = 
     7

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0

x = 
     9

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

Version History

Introduced in R2014b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

fractionlength

Fraction length of `quantizer` object

Syntax

```
fractionlength(q)
```

Description

`fractionlength(q)` returns the fraction length of `quantizer` object `q`.

Algorithms

For floating-point `quantizer` objects, $f = w - e - 1$, where w is the word length and e is the exponent length.

For fixed-point `quantizer` objects, f is part of the format $[w\ f]$.

Version History

Introduced before R2006a

See Also

`fi` | `numerictype` | `quantizer` | `wordlength`

fxpopt

Optimize data types of a system

Syntax

```
result = fxpopt(model, sud, options)
```

Description

`result = fxpopt(model, sud, options)` optimizes the data types in the model or subsystem specified by `sud` in the model, `model`, with additional options specified in the `fxpOptimizationOptions` object, `options`.

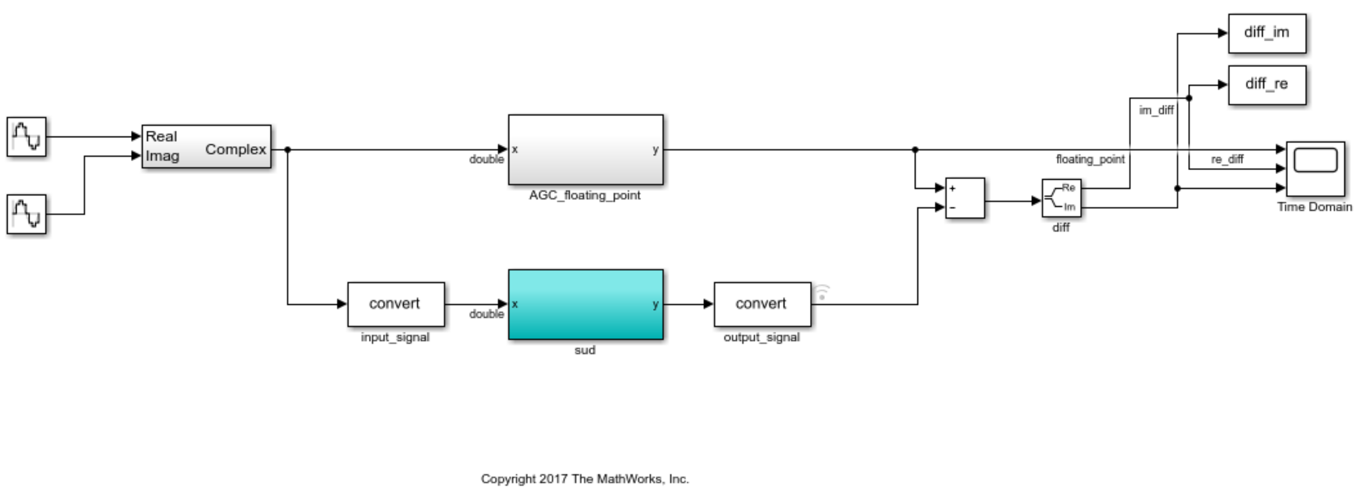
Examples

Optimize Fixed-Point Data Types

This example shows how to optimize the data types used by a system based on specified tolerances.

To begin, open the system for which you want to optimize the data types.

```
model = 'ex_auto_gain_controller';
sud = 'ex_auto_gain_controller/sud';
open_system(model)
```



Create an `fxpOptimizationOptions` object to define constraints and tolerances to meet your design goals. Set the `UseParallel` property of the `fxpOptimizationOptions` object to `true` to run iterations of the optimization in parallel. You can also specify word lengths to allow in your design through the `AllowableWordLengths` property.

```
opt = fxpOptimizationOptions('AllowableWordLengths', 10:24, 'UseParallel', true)
```

```
opt = 

  fxpOptimizationOptions with properties:

           MaxIterations: 50
                 MaxTime: 600
                Patience: 10
               Verbosity: High
    AllowableWordLengths: [10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
             UseParallel: 1

   Advanced Options
         AdvancedOptions: [1×1 struct]
```

Use the `addTolerance` method to define tolerances for the differences between the original behavior of the system, and the behavior using the optimized fixed-point data types.

```
tol = 10e-2;
addTolerance(opt, [model '/output_signal'], 1, 'AbsTol', tol);
```

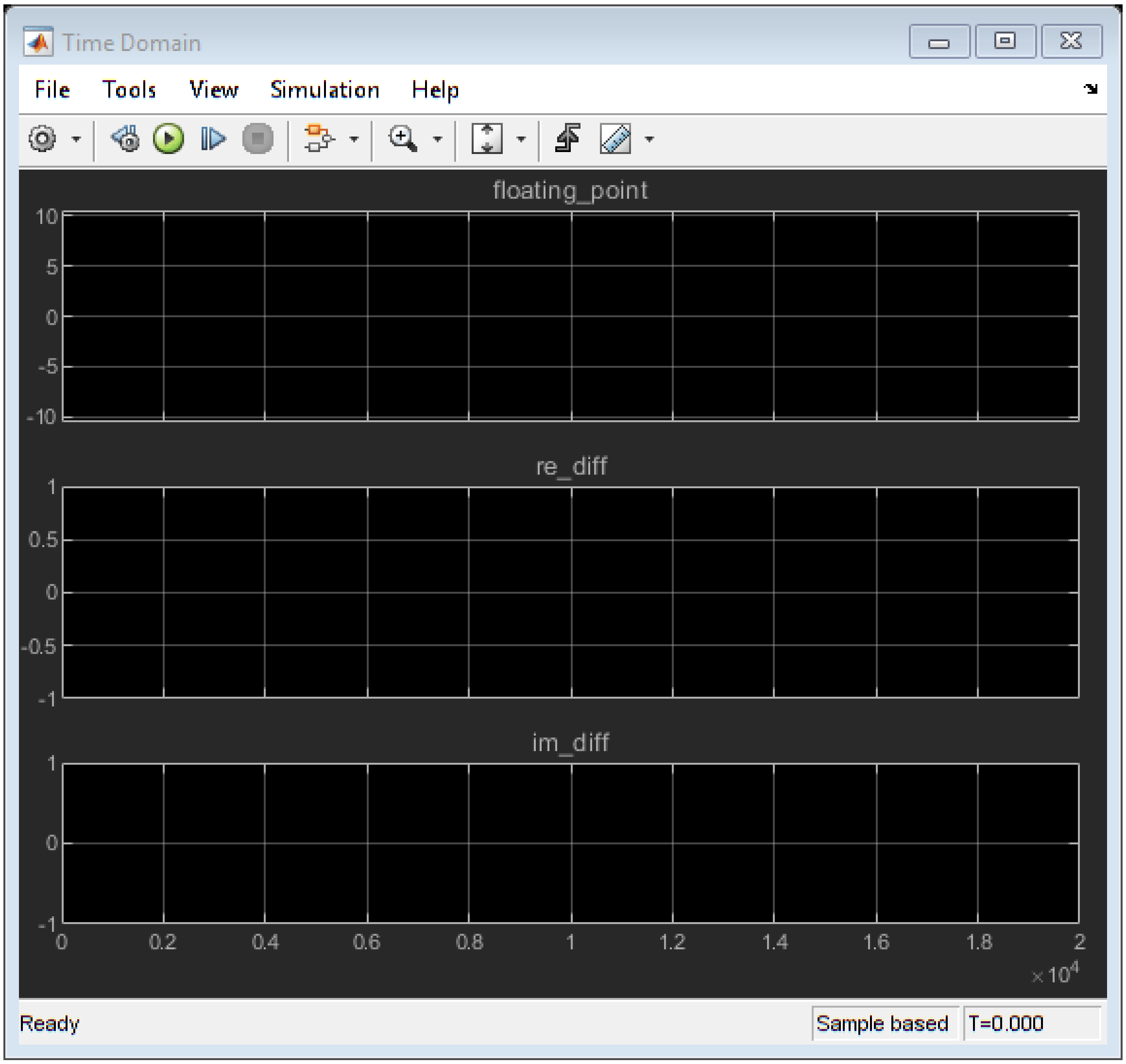
Use the `fxpopt` function to run the optimization. The software analyzes ranges of objects in your system under design and the constraints specified in the `fxpOptimizationOptions` object to apply heterogeneous data types to your system while minimizing total bit width.

```
result = fxpopt(model, sud, opt);
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 4).
    + Preprocessing
    + Modeling the optimization problem
        - Constructing decision variables
    + Running the optimization solver
Analyzing and transferring files to the workers ...done.
        - Evaluating new solution: cost 180, does not meet the tolerances.
        - Evaluating new solution: cost 198, does not meet the tolerances.
        - Evaluating new solution: cost 216, does not meet the tolerances.
        - Evaluating new solution: cost 234, does not meet the tolerances.
        - Evaluating new solution: cost 252, does not meet the tolerances.
        - Evaluating new solution: cost 270, does not meet the tolerances.
        - Evaluating new solution: cost 288, does not meet the tolerances.
        - Evaluating new solution: cost 306, meets the tolerances.
        - Evaluating new solution: cost 324, meets the tolerances.
        - Evaluating new solution: cost 342, meets the tolerances.
        - Evaluating new solution: cost 360, meets the tolerances.
        - Evaluating new solution: cost 378, meets the tolerances.
        - Evaluating new solution: cost 396, meets the tolerances.
        - Evaluating new solution: cost 414, meets the tolerances.
        - Evaluating new solution: cost 432, meets the tolerances.
        - Updated best found solution, cost: 306
        - Evaluating new solution: cost 304, meets the tolerances.
        - Evaluating new solution: cost 304, meets the tolerances.
        - Evaluating new solution: cost 301, meets the tolerances.
        - Evaluating new solution: cost 305, does not meet the tolerances.
        - Evaluating new solution: cost 305, meets the tolerances.
        - Evaluating new solution: cost 301, meets the tolerances.
        - Evaluating new solution: cost 299, meets the tolerances.
```

```
- Evaluating new solution: cost 299, meets the tolerances.
- Evaluating new solution: cost 296, meets the tolerances.
- Evaluating new solution: cost 299, meets the tolerances.
- Evaluating new solution: cost 291, meets the tolerances.
- Evaluating new solution: cost 296, does not meet the tolerances.
- Evaluating new solution: cost 299, meets the tolerances.
- Evaluating new solution: cost 300, meets the tolerances.
- Evaluating new solution: cost 296, does not meet the tolerances.
- Evaluating new solution: cost 301, meets the tolerances.
- Evaluating new solution: cost 303, meets the tolerances.
- Evaluating new solution: cost 299, meets the tolerances.
- Evaluating new solution: cost 304, does not meet the tolerances.
- Evaluating new solution: cost 300, meets the tolerances.
- Updated best found solution, cost: 304
- Updated best found solution, cost: 301
- Updated best found solution, cost: 299
- Updated best found solution, cost: 296
- Updated best found solution, cost: 291
- Evaluating new solution: cost 280, meets the tolerances.
- Evaluating new solution: cost 287, meets the tolerances.
- Evaluating new solution: cost 288, does not meet the tolerances.
- Evaluating new solution: cost 287, does not meet the tolerances.
- Evaluating new solution: cost 283, meets the tolerances.
- Evaluating new solution: cost 283, does not meet the tolerances.
- Evaluating new solution: cost 262, does not meet the tolerances.
- Evaluating new solution: cost 283, does not meet the tolerances.
- Evaluating new solution: cost 282, does not meet the tolerances.
- Evaluating new solution: cost 288, meets the tolerances.
- Evaluating new solution: cost 289, meets the tolerances.
- Evaluating new solution: cost 288, meets the tolerances.
- Evaluating new solution: cost 290, meets the tolerances.
- Evaluating new solution: cost 281, does not meet the tolerances.
- Evaluating new solution: cost 286, does not meet the tolerances.
- Evaluating new solution: cost 287, meets the tolerances.
- Evaluating new solution: cost 284, meets the tolerances.
- Evaluating new solution: cost 282, meets the tolerances.
- Evaluating new solution: cost 285, does not meet the tolerances.
- Evaluating new solution: cost 277, meets the tolerances.
- Updated best found solution, cost: 280
- Updated best found solution, cost: 277
- Evaluating new solution: cost 272, meets the tolerances.
- Evaluating new solution: cost 266, meets the tolerances.
- Evaluating new solution: cost 269, meets the tolerances.
- Evaluating new solution: cost 271, does not meet the tolerances.
- Evaluating new solution: cost 274, meets the tolerances.
- Evaluating new solution: cost 275, meets the tolerances.
- Evaluating new solution: cost 274, does not meet the tolerances.
- Evaluating new solution: cost 275, meets the tolerances.
- Evaluating new solution: cost 276, does not meet the tolerances.
- Evaluating new solution: cost 271, meets the tolerances.
- Evaluating new solution: cost 267, meets the tolerances.
- Evaluating new solution: cost 270, meets the tolerances.
- Evaluating new solution: cost 272, meets the tolerances.
- Evaluating new solution: cost 264, does not meet the tolerances.
- Evaluating new solution: cost 265, does not meet the tolerances.
- Evaluating new solution: cost 269, meets the tolerances.
- Evaluating new solution: cost 270, meets the tolerances.
- Evaluating new solution: cost 269, meets the tolerances.
```

```
    - Evaluating new solution: cost 276, meets the tolerances.
    - Evaluating new solution: cost 274, meets the tolerances.
    - Updated best found solution, cost: 272
    - Updated best found solution, cost: 266
+ Optimization has finished.
    - Neighborhood search complete.
    - Maximum number of iterations completed.
+ Fixed-point implementation that met the tolerances found.
    - Total cost: 266
    - Maximum absolute difference: 0.087035
    - Use the explore method of the result to explore the implementation.
```



Use the `explore` method of the `OptimizationResult` object, `result`, to launch Simulation Data Inspector and explore the design containing the smallest total number of bits while maintaining the numeric tolerances specified in the `opt` object.

```
explore(result);
```

You can revert your model back to its original state using the `revert` method of the `OptimizationResult` object.

```
revert(result);
```

Input Arguments

`model` — Model containing system under design, `sud`
character vector

Name of the model containing the system that you want to optimize.

Data Types: `char`

`sud` — Model or subsystem whose data types you want to optimize
character vector

Model or subsystem whose data types you want to optimize, specified as a character vector containing the path to the system.

Data Types: `char`

`options` — Additional optimization options
`fxpOptimizationOptions` object

`fxpOptimizationOptions` object specifying additional options to use during the data type optimization process.

Output Arguments

`result` — Object containing the optimized design
`OptimizationResult` object

Result of the optimization, returned as an `OptimizationResult` object. Use the `explore` method of the object to open the Simulation Data Inspector and view the behavior of the optimized system. You can also explore other solutions found during the optimization that may or may not meet the constraints specified in the `fxpOptimizationOptions` object, `options`.

Version History

Introduced in R2018a

See Also

Classes
`fxpOptimizationOptions` | `OptimizationResult` | `OptimizationSolution`

Functions
`addTolerance` | `showTolerances` | `explore`

Topics
“Optimize Fixed-Point Data Types for a System”

fxptdlg

Open the Fixed-Point Tool

Syntax

`fxptdlg(system_name)`

Description

`fxptdlg(system_name)` opens the Fixed-Point Tool for the Simulink model or subsystem specified by `system_name`.

You can also access this tool by the following methods:

- From the **Apps** tab, under **Code Generation** click **Fixed-Point Tool**.
- From a subsystem context (right-click) menu, select **Fixed-Point Tool**.

Examples

Open the Fixed-Point Tool from the Command Line

Open a Simulink model.

```
open_system('fxpdemo_feedback')
```

Open the Fixed-Point Tool with the `Controller` subsystem selected as the system under design.

```
fxptdlg('fxpdemo_feedback/Controller')
```

Override Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with the Fixed-Point Designer software. However, even if you do not have Fixed-Point Designer software, you can configure data type override settings to simulate a model that specifies fixed-point data types. In this mode, the Simulink software temporarily overrides fixed-point data types with floating-point data types when simulating the model.

Note that if you use fi on page 4-387 objects or embedded numeric data types in your model or workspace, you might introduce fixed-point data types into your model. You can set fipref on page 4-387 to prevent the checkout of a Fixed-Point Designer license.

To simulate a model without using Fixed-Point Designer:

Enter the following at the command line.

```
set_param(gcs, 'DataTypeOverride', 'Double',...
 'DataTypeOverrideAppliesTo','AllNumericTypes',...
 'MinMaxOverflowLogging','ForceOff')
```

If you use `fi` objects or embedded numeric data types in your model, set the `fipref` `DataTypeOverride` property to `TrueDoubles` or `TrueSingles` (to be consistent with the model-wide data type override setting) and the `DataTypeOverrideAppliesTo` property to `All numeric types`.

For example, at the MATLAB command line, enter:

```
p = fipref('DataTypeOverride', 'TrueDoubles', ...
       'DataTypeOverrideAppliesTo', 'AllNumericTypes');
```

Input Arguments

`system_name` — Model or subsystem to analyze or convert
top-level model of current system

Model or subsystem to analyze or convert in the Fixed-Point Tool.

Data Types: `string`

Version History

Introduced before R2006a

See Also

Fixed-Point Tool

ge, >=

Package: embedded

Determine whether real-world value of one array is greater than or equal to another

Syntax

```
A >= B
ge(A,B)
```

Description

A >= B returns a logical array with elements set to logical 1 (true) where the real-world values of A is greater than or equal to B, when A or B is a fi object. Otherwise, the element is logical 0 (false). The test compares only the real part of numeric arrays.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to a fixed-point type that preserves the relative *order* of the value with respect to the value in the fixed-point fi object.

ge(A,B) is an alternate way to execute A >= B, but is rarely used.

Examples

Compare Two fi Objects

Use the ge function to determine whether the real-world value of one fi object is greater than or equal to another.

```
a = fi(pi);
b = fi(pi, 1, 32);
b >= a
```

```
ans = logical
   0
```

Input a has a 16-bit word length, while input b has a 32-bit word length. The ge function returns 0 because after quantization, the value of a is slightly greater than that of b.

Compare a Double to a fi Object

When comparing a double to a fi object, the floating-point double is cast to a type that preserves the relative *order* of the value with respect to the value in the fixed-point fi object. This behavior allows relational operations to work between fi objects and floating-point constants without introducing floating-point values in generated code.

```
a = fi(pi);
b = pi;
ge(a,b)
```

```
ans =

  logical

   1
```

Input Arguments

A,B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs `A` and `B` must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators

Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for `fi` `ge`, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

R2022a: Improved accuracy in comparing `fi` objects and floating-point numbers using relational operators

Behavior changed in R2022a

In previous releases, when comparing a single or double to a `fi` object, the floating-point value was cast to the same word length and signedness of the `fi` object. This could lead to incorrect results. For example,

```
fi(0,0,8) > [-1,10]
```

```
ans =

  1×2 logical array

   0   0
```

```
fi(65534)
fi(65534.25) == 65534.25

ans =

       65534

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: -1

ans =

  logical

   1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   1   0
```

Note that the updated algorithm may produce subtle, but accurate, results. For example:

```
fi(pi) == pi

ans =

  logical

   0
```

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest wordlength growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

eq | gt | le | lt | ne

get

Property values of object

Syntax

```
value = get(o,'propertyname')
structure = get(o)
```

Description

`value = get(o,'propertyname')` returns the property value of the property `'propertyname'` for the object `o`. If you replace `'propertyname'` by a cell array of a vector of strings containing property names, `get` returns a cell array of a vector of corresponding values.

`structure = get(o)` returns a structure containing the properties and states of object `o`.

`o` can be a `fi`, `fimath`, `fipref`, `numerictype`, or `quantizer` object.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The syntax `structure = get(o)` is not supported.

See Also

`set`

getlsb

Package: `embedded`

Least significant bit

Syntax

```
c = getlsb(a)
```

Description

`c = getlsb(a)` returns the value of the least significant bit in `a`.

Examples

Find Least-Significant Bit in `fi` Object

Use `getlsb` to find the least-significnat bit in the `fi` object `a`.

```
a = fi(-26, 1, 6, 0);
c = getlsb(a)
```

```
c = 
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

You can verify that the least sigificant bit in the `fi` object `a` is `0` by looking at the binary representation of `a`.

```
disp(bin(a))
```

```
100110
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array. `getlsb` only supports `fi` object with fixed-point data types.

Data Types: `fi`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitand` | `bitandreduce` | `bitconcat` | `bitget` | `bitor` | `bitorreduce` | `bitset` | `bitxor` | `bitxorreduce` | `getmsb`

getmsb

Package: embedded

Most significant bit

Syntax

```
c = getmsb(a)
```

Description

`c = getmsb(a)` returns the value of the most-significant bit in `a`.

Examples

Find Most-Significant Bit in fi Object

Use `getmsb` to find the most-significant bit in the `fi` object `a`.

```
a = fi(-26, 1, 6, 0);
c = getmsb(a)
```

```
c = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

You can verify that the most significant bit in the `fi` object `a` is `1` by looking at the binary representation of `a`.

```
disp(bin(a))
```

```
100110
```

Input Arguments

a — Input fi object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array. `getmsb` only supports `fi` object with fixed-point data types.

Data Types: `fi`

Version History

Introduced in R2007b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`bitand` | `bitandreduce` | `bitconcat` | `bitget` | `bitor` | `bitorreduce` | `bitset` | `bitxor` | `bitxorreduce` | `getlsb`

globalfimath

Configure global fimath and return handle object

Syntax

```
G = globalfimath
G = globalfimath('PropertyName1',PropertyValue1,...)
G = globalfimath(f)
```

Description

`G = globalfimath` returns a handle object to the global fimath. The global fimath has identical properties to a `fimath` object but applies globally.

`G = globalfimath('PropertyName1',PropertyValue1,...)` sets the global fimath using the named properties and their corresponding values. Properties that you do not specify in this syntax are automatically set to that of the current global fimath.

`G = globalfimath(f)` sets the properties of the global fimath to match those of the input `fimath` object *f*, and returns a handle object to it.

Unless, in a previous release, you used the `saveglobalfimathpref` function to save global fimath settings to your MATLAB preferences, the global fimath properties you set with the `globalfimath` function apply only to your current MATLAB session. It is best practice to remove global fimath from the MATLAB preferences so that you start each MATLAB session using the default `fimath` settings. To remove the global fimath, use the `removeglobalfimathpref` function.

Examples

Modifying globalfimath

Use the globalfimath function to set, change, and reset the global fimath.

Create a fimath object and use it as the global fimath.

```
G = globalfimath('RoundMode','Floor','OverflowMode','Wrap')
```

```
G = 
        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Create another fimath object using the new default.

```
F1 = fimath
```

```
F1 = 
        RoundingMethod: Floor
        OverflowAction: Wrap
```

```
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Create a fi object, A, associated with the global fimath.

```
A = fi(pi)
```

```
A = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Now set the "SumMode" property of the global fimath to "KeepMSB" and retain all the other property values of the current global fimath.

```
G = globalfimath('SumMode','KeepMSB')
```

```
G = 
        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: KeepMSB
         SumWordLength: 32
         CastBeforeSum: true
```

Change the global fimath by directly interacting with the handle object G.

```
G.ProductMode = 'SpecifyPrecision'
```

```
G = 
        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: SpecifyPrecision
     ProductWordLength: 32
 ProductFractionLength: 30
               SumMode: KeepMSB
         SumWordLength: 32
         CastBeforeSum: true
```

Reset the global fimath to the factory default by calling the reset method on G. This is equivalent to using the resetglobalfimath function.

```
reset(G);
G
```

```
G = 
        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Tips

If you always use the same `fimath` settings and you are not sharing code with other people, using the `globalfimath` function is a quick, convenient method to configure these settings. However, if you share the code with other people or if you use the `fiaccel` function to accelerate the algorithm or you generate C code for your algorithm, consider the following alternatives.

Goal	Issue Using globalfimath	Solution
Share code	If you share code with someone who is using different global `fimath` settings, they might see different results.	Separate the `fimath` properties from your algorithm by using types tables. For more information, see “Separate Data Type Definitions from Algorithm”.
Accelerate your algorithm using `fiaccel` or generate C code from your MATLAB algorithm using `codegen`	You cannot use `globalfimath` within that algorithm. If you generate code with one `globalfimath` setting and run it with a different `globalfimath` setting, results might vary. For more information, see Specifying Default fimath Values for MEX Functions.	Use types tables in the algorithm from which you want to generate code. This insulates you from the global settings and makes the code portable. For more information, see “Separate Data Type Definitions from Algorithm”.

Version History

Introduced in R2010a

See Also

`fimath` | `codegen` | `fiaccel` | `removeglobalfimathpref` | `resetglobalfimath`

gt

Package: `embedded`

Determine whether real-world value of one array is greater than another

Syntax

```
A > B
gt(A,B)
```

Description

`A > B` returns a logical array with elements set to logical `1` (`true`) where the real-world values of `A` is greater than `B`, when `A` or `B` is a `fi` object. Otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to a fixed-point type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object.

`gt(A,B)` is an alternate way to execute `A > B`, but is rarely used.

Examples

Compare Two `fi` Objects

Use the `gt` function to determine whether the real-world value of one `fi` object is greater than another.

```
a = fi(pi);
b = fi(pi, 1, 32);
a > b
```

```
ans = logical
   1
```

Input `a` has a 16-bit word length, while input `b` has a 32-bit word length. The `gt` function returns `1` because after quantization, the value of `a` is greater than that of `b`.

Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the floating-point double is cast to a type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

```
a = fi(pi);
b = pi;
gt(a,b)

ans =

  logical

   1
```

Input Arguments

A,B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Version History

Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators
Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for fi gt, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a try/catch block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

R2022a: Improved accuracy in comparing fi objects and floating-point numbers using relational operators
Behavior changed in R2022a

In previous releases, when comparing a single or double to a fi object, the floating-point value was cast to the same word length and signedness of the fi object. This could lead to incorrect results. For example,

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   0   0
```

```
fi(65534)
fi(65534.25) == 65534.25

ans = 

       65534

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: -1

ans =

  logical

   1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   1   0
```

Note that the updated algorithm may produce subtle, but accurate, results. For example:

```
fi(pi) == pi

ans =

  logical

   0
```

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest wordlength growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

eq | ge | le | lt | ne

half

Construct half-precision numeric object

Description

Use the `half` constructor to assign a half-precision data type to a number or variable. A half-precision data type occupies 16 bits of memory, but its floating-point representation enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size. For more information, see “Floating-Point Numbers” and “What is Half Precision?”.

For a list of functions that support code generation with half-precision inputs, see “Half Precision Code Generation Support”.

Creation

Syntax

`a = half(v)`

Description

`a = half(v)` converts the values in `v` to half-precision.

Input Arguments

v — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Object Functions

These functions are supported for simulation with half-precision inputs in MATLAB. MATLAB System object™ supports half-precision data type and MATLAB System block supports half-precision data type with real values. For a list of functions that support code generation with half-precision inputs, see “Half Precision Code Generation Support”.

Math and Arithmetic

abs	Absolute value and complex magnitude
acos	Inverse cosine in radians
acosh	Inverse hyperbolic cosine
asin	Inverse sine in radians
asinh	Inverse hyperbolic sine

atan	Inverse tangent in radians
atan2	Four-quadrant inverse tangent
atanh	Inverse hyperbolic tangent
ceil	Round toward positive infinity
conj	Complex conjugate
conv	Convolution and polynomial multiplication
conv2	2-D convolution
cos	Cosine of argument in radians
cosh	Hyperbolic cosine
cospi	Compute cos(X*pi) accurately
cumsum	Cumulative sum
dot	Dot product
exp	Exponential
expm1	Compute exp(X)-1 accurately for small X
fft	Fast Fourier transform
fft2	2-D fast Fourier transform
fftn	N-D fast Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fix	Round toward zero
floor	Round toward negative infinity
fma	Multiply and add using fused multiply add approach
hypot	Square root of sum of squares (hypotenuse)
ifft	Inverse fast Fourier transform
ifft2	2-D inverse fast Fourier transform
ifftn	Multidimensional inverse fast Fourier transform
ifftshift	Inverse zero-frequency shift
imag	Imaginary part of complex number
ldivide	Left array division
log	Natural logarithm
log10	Common logarithm (base 10)
log1p	Compute natural logarithm of 1+X accurately for small X
log2	Base 2 logarithm and floating-point number dissection
mean	Average or mean value of array
minus	Subtraction
mldivide	Solve systems of linear equations Ax = B for x
mod	Remainder after division (modulo operation)
mrdivide	Solve systems of linear equations xA = B for x
mtimes	Matrix multiplication
plus	Add numbers, append strings
pow10	Base 10 power and scale half-precision numbers
pow2	Base 2 exponentiation and scaling of floating-point numbers
power	Element-wise power
prod	Product of array elements
rdivide	Right array division
real	Real part of complex number
rem	Remainder after division
round	Round to nearest decimal or integer
rsqrt	Reciprocal square root
sign	Sign function (signum function)
sin	Sine of argument in radians
sinh	Hyperbolic sine
sinpi	Compute sin(X*pi) accurately

sqrt	Square root
sum	Sum of array elements
tan	Tangent of argument in radians
tanh	Hyperbolic tangent
times	Multiplication
uminus	Unary minus
uplus	Unary plus

Data Types

allfinite	Determine if all array elements are finite
anynan	Determine if any array element is NaN
cast	Convert variable to different data type
cell	Cell array
double	Double-precision arrays
eps	Floating-point relative accuracy
flintmax	Largest consecutive integer in floating-point format
Inf	Create array of all Inf values
int16	16-bit signed integer arrays
int32	32-bit signed integer arrays
int64	64-bit signed integer arrays
int8	8-bit signed integer arrays
isa	Determine if input has specified data type
isfloat	Determine whether input is floating-point data type
isinteger	Determine whether input is integer array
islogical	Determine if input is logical array
isnan	Determine which array elements are NaN
isnumeric	Determine whether input is numeric array
isobject	Determine if input is MATLAB object
isreal	Determine whether array uses complex storage
logical	Convert numeric values to logicals
NaN	Create array of all NaN values
realmax	Largest positive floating-point number
realmin	Smallest normalized floating-point number
single	Single-precision arrays
storedInteger	Stored integer value of fi object
typecast	Convert data type without changing underlying data
uint16	16-bit unsigned integer arrays
uint32	32-bit unsigned integer arrays
uint64	64-bit unsigned integer arrays
uint8	8-bit unsigned integer arrays

Relational and Logical Operators

all	Determine if all array elements are nonzero or true
and	Find logical AND
Short-Circuit AND	Logical AND with short-circuiting
any	Determine if any array elements are nonzero
eq	Determine equality
ge	Determine greater than or equal to
gt	Determine greater than
isequal	Determine array equality
isequaln	Determine array equality, treating NaN values as equal

le	Determine less than or equal to
lt	Determine less than
ne	Determine inequality
not	Find logical NOT
or	Find logical OR
Short-Circuit OR	Logical OR with short-circuiting

Array and Matrix Operations

cat	Concatenate arrays
chol	Cholesky factorization
circshift	Shift array circularly
colon	Vector creation, array subscripting, and for-loop iteration
complex	Create complex array
ctranspose	Complex conjugate transpose
empty	Create empty array of specified class
eye	Identity matrix
flip	Flip order of elements
fliplr	Flip array left to right
flipud	Flip array up to down
horzcat	Horizontal concatenation for heterogeneous arrays
iscolumn	Determine if input is column vector
isempty	Determine whether array is empty
isfinite	Determine which array elements are finite
isinf	Determine which array elements are infinite
ismatrix	Determine whether input is matrix
isrow	Determine if input is row vector
isscalar	Determine whether input is scalar
issorted	Determine if array is sorted
isvector	Determine whether input is vector
length	Length of largest array dimension
lu	LU matrix factorization
max	Maximum elements of array
min	Minimum elements of array
ndims	Number of array dimensions
numel	Number of array elements
ones	Create array of all ones
permute	Permute array dimensions
repelem	Repeat copies of array elements
repmat	Repeat copies of array
reshape	Reshape array
size	Array size
sort	Sort array elements
squeeze	Remove dimensions of length 1
transpose	Transpose vector or matrix
vertcat	Vertically concatenate for heterogeneous arrays
zeros	Create array of all zeros

Graphics

area	Area of 2-D alpha shape
bar	Bar graph
barh	Horizontal bar graph

fplot	Plot expression or function
line	Create primitive line
plot	2-D line plot
plot3	3-D point or line plot
plotmatrix	Scatter plot matrix
rgbplot	Plot colormap
scatter	Scatter plot
scatter3	3-D scatter plot
xlim	Set or query x-axis limits
ylim	Set or query y-axis limits
zlim	Set or query z-axis limits

Deep Learning

activations	Compute deep learning network layer activations
classify	Classify data using trained deep learning neural network
predict	Reconstruct the inputs using trained autoencoder
predictAndUpdateState	Predict responses using a trained recurrent neural network and update the network state

To display a list of supported functions, at the MATLAB Command Window, enter:

```
methods(half(1))
```

Examples

Convert Value to Half Precision

To cast a double-precision number to half precision, use the `half` function.

```
a = half(pi)
```

```
a = 

  half

    3.1406
```

You can also use the `half` function to cast an existing variable to half-precision.

```
v = single(magic(3))
```

```
v = 3x3 single matrix

     8     1     6
     3     5     7
     4     9     2
```

```
a = half(v)
```

```
a = 

  3x3 half matrix
```

```
8     1     6
3     5     7
4     9     2
```

Limitations

- Arithmetic operations which combine half-precision and logical types are not supported.
- For additional usage notes and limitations, see “Half Precision Code Generation Support”.

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- For a list of functions that support code generation with half-precision inputs and any associated limitations, see “Half Precision Code Generation Support”.
- If your target hardware does not have native support for half-precision, then `half` is used as a storage type, with arithmetic operations performed in single-precision.
- Some functions use `half` only as a storage type and the arithmetic is performed in single-precision, regardless of the target hardware.
- For deep learning code generation, half inputs are cast to single precision and computations are performed in single precision.
- In MATLAB, the `isobject` function returns true with a half-precision input. In generated code, this function returns false.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

- For a list of functions that support code generation with half-precision inputs and any associated limitations, see “Half Precision Code Generation Support”.
- CUDA® compute capability of 5.3 or higher is required for generating and executing code with half-precision data types.
- CUDA toolkit version of 10.0 or later is required for generating and executing code with half-precision data types.
- You must set the memory allocation (`malloc`) mode to `'Discrete'` for generating CUDA code.
- Half-precision complex data types are not supported for GPU code generation.
- If your target hardware does not have native support for half-precision, then `half` is used as a storage type, with arithmetic operations performed in single-precision.
- Some functions use `half` only as a storage type and the arithmetic is performed in single-precision, regardless of the target hardware.
- For deep learning code generation, half inputs are cast to single precision and computations are performed in single precision. To perform computations in half, set the library target to `'tensorrt'` and set the data type to `'FP16'` in `coder.DeepLearningConfig`.

- In MATLAB, the `isobject` function returns true with a half-precision input. In generated code, this function returns false.

See Also

`single` | `double`

Topics

"Half Precision Code Generation Support"
"Floating-Point Numbers"
"What is Half Precision?"
"Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type" (MATLAB Coder)
Edge Detection with Sobel Method in Half-Precision (GPU Coder)

hex

Package: `embedded`

Hexadecimal representation of stored integer of `fi` object

Syntax

`b = hex(a)`

Description

`b = hex(a)` returns the stored integer of `fi` object `a` in hexadecimal format as a character vector.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

Tip `hex` returns the hexadecimal representation of the stored integer of a `fi` object. To obtain the hexadecimal representation of the real-world value of a `fi` object, use `dec2hex`.

Examples

View Stored Integer of `fi` Object in Hexadecimal Format

Create a signed `fi` object with values `-1` and `1`, a word length of 8 bits, and a fraction length of 7 bits.

```
a = fi([-1 1], 1, 8, 7)
```

```
a =
   -1.0000    0.9922

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Find the hexadecimal representation of the stored integers of `fi` object `a`.

```
b = hex(a)
```

```
b =
'80   7f'
```

Write Hex Data to a File

This example shows how to write hexadecimal data from the MATLAB workspace into a text file.

Define your data and create a writable text file called `hexdata.txt`.

```
x = (0:15)'/16;
a = fi(x, 0, 16, 16);
h = fopen('hexdata.txt', 'w');
```

Use the `fprintf` function to write your data to the `hexdata.txt` file.

```
for k = 1:length(a)
    fprintf(h, '%s\n', hex(a(k)));
end

fclose(h);
```

To see the contents of the file you created, use the `type` function.

```
type hexdata.txt
```

```
0000
1000
2000
3000
4000
5000
6000
7000
8000
9000
a000
b000
c000
d000
e000
f000
```

Read Hex Data From a File

This example shows how to read hexadecimal data from a text file back into the MATLAB workspace.

Define your data, create a writable text file called `hexdata.txt`, and write your data to the `hexdata.txt` file.

```
x = (0:15)'/16;
a = fi(x, 0, 16, 16);
h = fopen('hexdata.txt', 'w');

for k = 1:length(a)
    fprintf(h, '%s\n', hex(a(k)));
end
```

```
fclose(h);
```

Open `hexdata.txt` for reading and read its contents into a workspace variable

```
h = fopen('hexdata.txt', 'r');

nextline = '';
str = '';

while ischar(nextline)
    nextline = fgetl(h);
    if ischar(nextline)
        str = [str; nextline];
    end
end

fclose(h);
```

Create a `fi` object with the correct scaling and assign it the hex values stored in the `str` variable.

```
b = fi([], 0, 16, 16);
b.hex = str
```

```
b =
         0
    0.0625
    0.1250
    0.1875
    0.2500
    0.3125
    0.3750
    0.4375
    0.5000
    0.5625
    0.6250
    0.6875
    0.7500
    0.8125
    0.8750
    0.9375

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 16
```

Input Arguments

a — Input array
`fi` object

Input array, specified as a `fi` object.

Data Types: `fi`

Version History

Introduced before R2006a

See Also

`bin` | `dec` | `storedInteger` | `oct` | `dec2hex` | `dec2base` | `dec2bin`

hex2num

Package: `embedded`

Convert hexadecimal string to number using `quantizer` object

Syntax

```
x = hex2num(q,h)
[x1,x2,…] = hex2num,(q,h1,h2,…)
```

Description

`x = hex2num(q,h)` converts hexadecimal character vector `h` to numeric matrix `x`.

The attributes of the numbers in `x` are specified by `quantizer` object `q`. When `h` is a cell array, `hex2num` returns `x` as a cell array of the same dimension containing numbers. For fixed-point hexadecimal representations, `hex2num` uses “Two's Complement” representation. For floating point, the representation is IEEE Standard 754 style.

When there are fewer hexadecimal digits than needed to represent the number, the fixed-point conversion zero-fills on the left. Floating-point conversion zero-fills on the right.

`hex2num` and `num2hex` are inverses of one another, with the distinction that `num2hex` returns the hexadecimal representations in a column.

Note This function uses a `quantizer` object to convert a hexadecimal string to a number. To convert IEEE hexadecimal format to a double-precision number without using a `quantizer` object, use the MATLAB `hex2num` function.

`[x1,x2,…] = hex2num,(q,h1,h2,…)` converts hexadecimal representations `h1,h2,…` to numeric matrices `x1,x2,…`.

Examples

Convert Hexadecimal Array to Numeric Array

This example returns all the 4-bit fixed-point two's complement numbers in fractional form.

```
q = quantizer([4 3]);
h = ['7  3  F  B'
     '6  2  E  A'
     '5  1  D  9'
     '4  0  C  8'];
x = hex2num(q,h)
```

```
x = 4×4

    0.8750    0.3750   -0.1250   -0.6250
```

```
    0.7500    0.2500   -0.2500   -0.7500
    0.6250    0.1250   -0.3750   -0.8750
    0.5000         0   -0.5000   -1.0000
```

Input Arguments

q — Input `quantizer` object
`quantizer` object

Input `quantizer` object, specified as a `quantizer` object.

h — Text representing hexadecimal numbers
character array

Text representing hexadecimal numbers, specified as a character array.

Data Types: `char`

`h1,h2,…` — Text representing hexadecimal numbers
character array

Text representing hexadecimal numbers, specified as a character array.

Data Types: `char`

Version History

Introduced before R2006a

See Also

`bin2num` | `num2bin` | `num2hex` | `num2int`

horzcat

Concatenate multiple fi objects horizontally

Syntax

```
C = horzcat(A,B)
C = horzcat(A1,A2,…An)
```

Description

`C = horzcat(A,B)` concatenates B horizontally to the end of A when A and B have compatible sizes (the lengths of the dimensions match except in the second dimension).

`C = horzcat(A1,A2,…An)` concatenates A1,A2,…,An horizontally.

`horzcat` is equivalent to using square brackets for horizontally concatenating arrays. For example, [A,B] or [A B] is equal to `horzcat(A,B)` when A and B are compatible arrays.

Note The fimath and numerictype properties of a concatenated matrix of fi objects, C, are taken from the leftmost fi object in the list A1,A2,…,An.

Input Arguments

A — First input
scalar | vector | matrix | multidimensional array

First input, specified as a scalar, vector, matrix, or multidimensional array.

B — Second input
scalar | vector | matrix | multidimensional array

Second input, specified as a scalar, vector, matrix, or multidimensional array.

The elements of B are concatenated to the end of the first input along the second dimension. The sizes of the input arguments must be compatible. For example, if the first input is a matrix of size 3-by-2, then B must have 3 rows.

A1,A2,…An — List of inputs
scalar | vector | matrix | multidimensional array

List of inputs, specified as a comma-separated list of elements to concatenate in the order they are specified.

Any number of matrices can be concatenated within one pair of brackets. Multidimensional arrays are horizontally concatenated along the second dimension.

The inputs must have compatible sizes. For example, if A1 is a column vector of length *m*, then the remaining inputs must each have *m* rows to concatenate horizontally.

Tips

- Horizontal and vertical concatenation can be combined together, as in `[1 2;3 4]`.
- The matrices in a concatenation expression can themselves be formed via a concatenation, as in `[a b;[c d]]`.
- `[A B;C]` is allowed if the number of rows of `A` equals the number of rows of `B` and if the number of columns of `A` plus the number of columns of `B` equals the number of columns of `C`.
- When concatenating an empty array to a nonempty array, `horzcat` omits the empty array in the output. For example,

```
horzcat(fi([1 2]),[])
```

```
ans = 

     1     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`vertcat`

innerprodintbits

Number of integer bits needed for fixed-point inner product

Syntax

`innerprodintbits(a,b)`

Description

`innerprodintbits(a,b)` computes the minimum number of integer bits necessary in the inner product of `a'*b` to guarantee that no overflows occur and to preserve best precision.

- `a` and `b` are `fi` vectors.
- The values of `a` are known.
- Only the numeric type of `b` is relevant. The values of `b` are ignored.

Examples

The primary use of this function is to determine the number of integer bits necessary in the output `Y` of an FIR filter that computes the inner product between constant coefficient row vector `B` and state column vector `Z`. For example,

```
for k=1:length(X);
  Z = [X(k);Z(1:end-1)];
  Y(k) = B * Z;
end
```

Algorithms

In general, an inner product grows `log2(n)` bits for vectors of length `n`. However, in the case of this function the vector `a` is known and its values do not change. This knowledge is used to compute the smallest number of integer bits that are necessary in the output to guarantee that no overflow will occur.

The largest gain occurs when the vector `b` has the same sign as the constant vector `a`. Therefore, the largest gain due to the vector `a` is `a*sign(a')`, which is equal to `sum(abs(a))`.

The overall number of integer bits necessary to guarantee that no overflow occurs in the inner product is computed by:

```
n = ceil(log2(sum(abs(a)))) + number of integer bits in b + 1 sign bit
```

The extra sign bit is only added if both `a` and `b` are signed and `b` attains its minimum. This prevents overflow in the event of (-1)*(-1).

Version History

Introduced before R2006a

int

Get stored integer value of a fi object

Syntax

```
i = int(a)
```

Description

`i = int(a)` returns the integer value of a `fi` object, stored in one of the built-in integer data types.

Examples

Get the Stored Integer Value of a fi Object

Create a `fi` object with default settings. Use the `int` function to get its stored integer value. The output is an `int16` because the input used the default word length of 16-bits.

```
a = fi(pi);
b = int(a)
```

```
b = int16
    25736
```

Create a `fi` object that uses a 20-bit word length and get the stored integer value of the `fi` object.

```
a = fi(pi,1,20);
b = int(a)
```

```
b = int32
    411775
```

The output is an `int32` to accommodate the larger input word length.

Input Arguments

a — Fixed-point numeric object
scalar | vector | matrix | multidimensional array

Fixed-point numeric object from which you want to get the stored integer value. The word length of the input determines the data type of the output.

Data Types: `fi`
Complex Number Support: Yes

Output Arguments

i — Stored integer value
scalar | vector | matrix | multidimensional array

Stored integer value of the input `fi` object, returned as one of the built-in integer data types. The word length of the input determines the data type of the output. The output has the same dimensions as the input.

Version History

Introduced in R2006a

See Also

Functions
`bin` | `hex` | `storedInteger` | `oct` | `sdec`

int8

Convert `fi` object to signed 8-bit integer

Syntax

`c = int8(a)`

Description

`c = int8(a)` returns the built-in `int8` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int8`.

Examples

This example shows the `int8` values of a fi object.

```
a = fi([-pi 0.1 pi],1,8);
c = int8(a)

c =

   -3    0    3
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`storedInteger` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

int16

Convert `fi` object to signed 16-bit integer

Syntax

`c = int16(a)`

Description

`c = int16(a)` returns the built-in `int16` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int16`.

Examples

This example shows the `int16` values of a fi object.

```
a = fi([-pi 0.1 pi],1,16);
c = int16(a)

c =

   -3    0    3
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`storedInteger` | `int8` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

int32

Convert `fi` object to signed 32-bit integer

Syntax

`c = int32(a)`

Description

`c = int32(a)` returns the built-in `int32` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `int32`.

Examples

This example shows the `int32` values of a fi object.

```
a = fi([-pi 0.1 pi],1,32);
c = int32(a)

c =

   -3    0    3
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`storedInteger` | `int8` | `int16` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

int64

Convert fi object to signed 64-bit integer

Syntax

```
c = int64(a)
```

Description

c = int64(a) returns the built-in int64 value of fi object a, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an int64.

Examples

This example shows the int64 values of a fi object.

```
a = fi([-pi 0.1 pi],1,64);
c = int64(a)

c =

   -3    0    3
```

Version History

Introduced in R2008b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

storedInteger | int8 | int16 | int32 | uint8 | uint16 | uint32 | uint64

intmax

Largest positive stored integer value representable by `numerictype` of `fi` object

Syntax

`x = intmax(a)`

Description

`x = intmax(a)` returns the largest positive stored integer value representable by the `numerictype` of `a`.

Examples

```
a = fi(pi, true, 16, 12);
x = intmax(a)
```

```
x =

       32767

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

Version History

Introduced before R2006a

See Also

`eps` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `stripscaling` | `upperbound`

intmin

Smallest stored integer value representable by `numerictype` of `fi` object

Syntax

`x = intmin(a)`

Description

`x = intmin(a)` returns the smallest stored integer value representable by the `numerictype` of `a`.

Examples

```
a = fi(pi, true, 16, 12);
x = intmin(a)

x =

      -32768

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 0
```

Version History

Introduced before R2006a

See Also

`eps` | `intmax` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin` | `stripscaling` | `upperbound`

isboolean

Determine whether input is Boolean

Syntax

```
tf = isboolean(a)
tf = isboolean(T)
```

Description

`tf = isboolean(a)` returns `1` (`true`) when the `DataType` property of `fi` object `a` is `Boolean`. Otherwise, it returns `0` (`false`).

`tf = isboolean(T)` returns `1` (`true`) when the `DataType` property of `numerictype` object `T` is `Boolean`. Otherwise, it returns `0` (`false`).

Examples

Determine Whether `fi` Object is a Boolean

Create a `fi` object and determine if its data type is `Boolean`.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isboolean(a)
```

```
tf = logical
   0
```

```
a = fi(pi,'DataType','Boolean')
```

```
a = 
   1

          DataTypeMode: Boolean
```

```
tf = isboolean(a)
```

```
tf = logical
   1
```

Determine Whether numerictype Object is a Boolean

Create a `numerictype` object and determine if its data type is `Boolean`.

```
T = numerictype
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isboolean(T)
```

```
tf = logical
   0
```

```
T = numerictype('Boolean')
```

```
T =


          DataTypeMode: Boolean
```

```
tf = isboolean(T)
```

```
tf = logical
   1
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

Version History
Introduced in R2008a

See Also
`isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isdouble

Determine whether input is double-precision data type

Syntax

```
tf = isdouble(a)
tf = isdouble(T)
```

Description

`tf = isdouble(a)` returns `1` (`true`) when the `DataType` property of `fi` object `a` is double. Otherwise, it returns `0` (`false`).

`tf = isdouble(T)` returns `1` (`true`) when the `DataType` property of `numerictype` object `T` is `double`. Otherwise, it returns `0` (`false`).

Examples

Determine Whether `fi` Object is a `double`

Create a `fi` object and determine if its data type is `double`.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isdouble(a)
```

```
tf = logical
   0
```

```
a = fi(pi,'DataType','double')
```

```
a = 
    3.1416

          DataTypeMode: Double
```

```
tf = isdouble(a)
```

```
tf = logical
   1
```

Determine Whether numerictype Object is a double

Create a `numerictype` object and determine if its data type is `double`.

```
T = numerictype
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isdouble(T)
```

```
tf = logical
   0
```

```
T = numerictype('Double')
```

```
T =


          DataTypeMode: Double
```

```
tf = isdouble(T)
```

```
tf = logical
   1
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2008a

See Also

`isboolean` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isequal

Determine whether real-world values of two `fi` objects are equal, or determine whether properties of two `fimath`, `numerictype`, or `quantizer` objects are equal

Syntax

```
y = isequal(a,b,…)
y = isequal(F,G,…)
y = isequal(T,U,…)
y = isequal(q,r,…)
```

Description

`y = isequal(a,b,…)` returns logical `1` (`true`) if all the `fi` object inputs have the same real-world value. Otherwise, it returns logical `0` (`false`).

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to the same word length and signedness as the `fi` object, with best-precision scaling.

`y = isequal(F,G,…)` returns logical `1` (`true`) if all the `fimath` object inputs have the same properties. Otherwise, it returns logical `0` (`false`).

`y = isequal(T,U,…)` returns logical `1` (`true`) if all the `numerictype` object inputs have the same properties. Otherwise, it returns logical `0` (`false`).

`y = isequal(q,r,…)` returns logical `1` (`true`) if all the `quantizer` object inputs have the same properties. Otherwise, it returns logical `0` (`false`).

Examples

Compare Two `fi` Objects

Use the `isequal` function to determine if two `fi` objects have the same real-world value.

```
format long
a = fi(pi)
```

```
a = 
   3.141601562500000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
b = fi(pi,1,32)
```

```
b = 
   3.141592653468251
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 29
```

```
y = isequal(a,b)
```

```
y = logical
   0
```

Input a has a 16-bit word length, while input b has a 32-bit word length. The isequal function returns 0 because the two fi objects do not have the same real-world value.

Compare a Double to a fi Object

When comparing a double to a fi object, the double is cast to the same word length and signedness of the fi object.

```
a = fi(pi);
b = pi;
y = isequal(a,b)
```

```
y = logical
   1
```

The isequal function casts b to the same word length as a, and returns 1. This behavior allows relational operations to work between fi objects and floating-point constants without introducing floating-point values in generated code.

Compare Two fimath Objects

Use the isequal function to determine if two fimath objects have the same properties.

```
F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent');
G = fimath('RoundingMethod','Convergent','ProductMode','FullPrecision');
y = isequal(F,G)
```

```
y = logical
   1
```

Compare Two numerictype Objects

Use the isequal function to determine if two numerictype objects have the same properties.

```
T = numerictype;
U = numerictype(true, 16, 15);
y = isequal(T,U)
```

```
y = logical
   1
```

Compare Two quantizer Objects

Use the `isequal` function to determine if two `quantizer` objects have the same properties.

```
q = quantizer('fixed', [5 4]);
r = quantizer('fixed', 'floor', 'saturate', [5 4]);
y = isequal(q,r)
```

```
y = logical
   1
```

Input Arguments

a,b,… — fi objects to be compared
scalar | vector | matrix | multidimensional array

`fi` objects to be compared, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`
Complex Number Support: Yes

F,G,… — fimath objects to be compared
`fimath` object

`fimath` objects to be compared.

T,U,… — numerictype objects to be compared
scalar | vector | matrix | multidimensional array

`numerictype` objects to be compared, specified as a scalar, vector, matrix, or multidimensional array.

q,r,… — quantizer objects to be compared
`quantizer` object

`quantizer` objects to be compared.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eq` | `fi` | `fimath` | `ispropequal` | `numerictype` | `quantizer`

isequivalent

Determine if two `numerictype` objects have equivalent properties

Syntax

```
y = isequivalent (T1, T2)
```

Description

`y = isequivalent (T1, T2)` determines whether the `numerictype` object inputs have equivalent properties and returns a logical `1` (`true`) or `0` (`false`). Two `numerictype` objects are equivalent if they describe the same data type.

Examples

Compare two numerictype objects

Use `isequivalent` to determine if two `numerictype` objects have the same data type.

```
T1 = numerictype(1, 16, 2^-12, 0)
```

```
T1 =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^-12
                  Bias: 0
```

```
T2 = numerictype(1, 16, 12)
```

```
T2 =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

```
isequivalent(T1,T2)
```

```
ans = logical
   1
```

Although the Data Type Mode is different for `T1` and `T2`, the function returns `1` (`true`) because the two objects have the same data type.

Input Arguments

T1, T2 — Inputs to be compared
numerictype objects

Inputs to be compared, specified as `numerictype` objects.

Version History
Introduced in R2014a

See Also
isequal | ispropequal | eq

isequaln

Package: embedded

Determine equality of fixed-point arrays, treating NaN values as equal

Syntax

```
tf = isequaln(A,B)
tf = isequaln(A1,A2,…,An)
```

Description

`tf = isequaln(A,B)` returns logical `1` (`true`) if `A` and `B` are equivalent; otherwise, it returns logical `0` (`false`). Arrays are considered equivalent if they are the same size and are numerically equal. `NaN` (Not a Number) values are considered to be equal to other such values.

Numeric data types and structure field order need not match to be considered equivalent.

`isequaln` recursively compares the contents of cell arrays and structures. If all elements of a cell array or structure are numerically equal, `isequaln` returns logical `1` (`true`).

`tf = isequaln(A1,A2,…,An)` returns logical `1` (`true`) if all the inputs are equivalent.

Examples

Compare Two Numeric Matrices

Create two numeric matrices and compare them for equality.

```
A = fi(zeros(3,3)+1e-4,1,16,15);
B = fi(zeros(3,3),1,16,15);
tf = isequaln(A,B)
```

```
tf = logical
   0
```

The function returns logical `0` (`false`) because the matrices differ by a small amount and are not *exactly* equal.

If the two matrices differ by an amount that is smaller than the precision representable by the fixed-point data type, the function returns logical 1 (true).

```
A = fi(zeros(3,3)+1e-5,1,16,15);
B = fi(zeros(3,3),1,16,15);
tf = isequaln(A,B)
```

```
tf = logical
   1
```

Two matrices can be considered numerically equivalent when the inputs are of different data types.

```
A = fi(zeros(3,3),1,16,15);
B = single(zeros(3,3));
tf = isequaln(A,B)
```

```
tf = logical
   1
```

Input Arguments

A,B — Inputs to be compared (as separate arguments)
arrays

Inputs to be compared, specified as arrays.

Data Types: `fi`
Complex Number Support: Yes

A1,A2,…,An — Series of inputs to be compared (as separate inputs)
arrays

Series of inputs to be compared, specified as arrays.

Data Types: `fi`
Complex Number Support: Yes

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`isequal` | `eq`

isfi

Determine whether variable is `fi` object

Syntax

```
tf = isfi(a)
```

Description

`tf = isfi(a)` returns `1` (`true`) if `a` is a `fi` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Variable is a `fi` Object

Create a variable and determine whether it is a `fi` object.

```
a = fi(pi);
tf = isfi(a)
```

```
tf = logical
   1
```

```
b = single([1 2 3 4]);
tf = isfi(b)
```

```
tf = logical
   0
```

Input Arguments

a — Input array
array

Input array.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Avoid using the `isfi` function in code that you intend to convert using the automated workflow. The value returned by `isfi` in the fixed-point code might differ from the value returned in the original MATLAB algorithm. The behavior of the fixed-point code might differ from the behavior of the original algorithm.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`fi` | `isfimath` | `isfipref` | `isnumerictype` | `isquantizer`

isfimath

Determine whether variable is `fimath` object

Syntax

`tf = isfimath(F)`

Description

`tf = isfimath(F)` returns `1` (`true`) if `F` is a `fimath` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Variable is a `fimath` Object

Create a variable and determine whether it is a `fimath` object

```
F = fimath;
tf = isfimath(F)
```

```
tf = logical
   1
```

```
T = numerictype;
tf = isfimath(T)
```

```
tf = logical
   0
```

```
A = fi([1 2 3 4]);
tf = isfimath(A)
```

```
tf = logical
   0
```

Input Arguments

F — Input array
array

Input array.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`fimath` | `isfi` | `isfipref` | `isnumerictype` | `isquantizer`

isfimathlocal

Determine whether `fi` object has local `fimath`

Syntax

`tf = isfimathlocal(a)`

Description

`tf = isfimathlocal(a)` returns `1` (`true`) if the `fi` object `a` has a local `fimath` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether `fi` Object has Local `fimath`

Create a `fi` object and determine whether it has local `fimath`.

```
F = fimath;
a = fi(pi);
b = fi(pi,F);

tf_a = isfimathlocal(a)
```

```
tf_a = logical
   0
```

```
tf_b = isfimathlocal(b)
```

```
tf_b = logical
   1
```

Input Arguments

a — Input array
array

Input array.

Data Types: `fi`

Version History

Introduced in R2009b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`fimath` | `isfi` | `isfipref` | `isnumerictype` | `isquantizer` | `removefimath` | `sfi` | `ufi`

isfipref

Determine whether input is `fipref` object

Syntax

`tf = isfipref(P)`

Description

`tf = isfipref(P)` returns `1` (`true`) if `P` is a `fipref` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input is a `fipref` Object

Create a variable and determine whether it is a `fipref` object.

```
P = fipref;
tf = isfipref(P)
```

```
tf = logical
   1
```

```
F = fimath;
tf = isfipref(F)
```

```
tf = logical
   0
```

Input Arguments

P — Input array
array

Input array.

Version History

Introduced in R2008a

See Also

`fipref` | `isfi` | `isfimath` | `isnumerictype` | `isquantizer`

isfixed

Determine whether input is fixed-point data type

Syntax

```
tf = isfixed(a)
tf = isfixed(T)
tf = isfixed(q)
```

Description

`tf = isfixed(a)` returns `1` (`true`) when the `DataType` property of `fi` object `a` is `Fixed`. Otherwise, it returns `0` (`false`).

`tf = isfixed(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `Fixed`. Otherwise, it returns `0` (`false`).

`tf = isfixed(q)` returns `1` when `q` is a fixed-point `quantizer` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input is a Fixed-Point Data Type

Create a `fi` object and determine whether it is a fixed-point data type.

```
a = fi([pi pi/2])
```

```
a = 
    3.1416    1.5708

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isfixed(a)
```

```
tf = logical
   1
```

Create a `numerictype` object and determine whether it is a fixed-point data type.

```
T = numerictype('Double')
```

```
T = 


          DataTypeMode: Double
```

```
tf = isfixed(T)
```

```
tf = logical
   0
```

Create a `quantizer` object and determine whether it is a fixed-point data type.

```
q = quantizer('mode','single')
```

```
q =


        DataMode = single
          Format = [32  8]
```

```
tf = isfixed(q)
```

```
tf = logical
   0
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

q — Input `quantizer` object
scalar

Input `quantizer` object, specified as a scalar.

Version History

Introduced in R2008a

See Also

`isboolean` | `isdouble` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isfloat

Determine whether input is floating-point data type

Syntax

```
y = isfloat(a)
y = isfloat(T)
y = isfloat(q)
```

Description

`y = isfloat(a)` returns `1` when the `DataType` property of `fi` object `a` is `single`, or `double`, and `0` otherwise.

`y = isfloat(T)` returns `1` when the `DataType` property of `numerictype` object `T` is `single`, or `double`, and `0` otherwise.

`y = isfloat(q)` returns `1` when `q` is a floating-point `quantizer`, and `0` otherwise.

Version History

Introduced in R2008a

See Also

`isboolean` | `isdouble` | `isfixed` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isnumerictype

Determine whether input is `numerictype` object

Syntax

```
tf = isnumerictype(T)
```

Description

`tf = isnumerictype(T)` returns `1` (`true`) if `T` is a `numerictype` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input is a `numerictype` Object

Create a variable and determine whether it is a `numerictype` object.

```
T = numerictype;
tf = isnumerictype(T)
```

```
tf = logical
   1
```

```
q = quantizer;
tf = isnumerictype(q)
```

```
tf = logical
   0
```

Input Arguments

T — Input array
array

Input array.

Version History
Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`isfi` | `isfimath` | `isfipref` | `isquantizer` | `numerictype`

ispropequal

Determine whether properties of two `fi` objects are equal

Syntax

```
tf = ispropequal(a,b)
```

Description

`tf = ispropequal(a,b)` returns `1` (`true`) if `a` and `b` are both `fi` objects and have the same properties. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Properties of Two `fi` Objects are Equal

Create two `fi` objects and determine whether they have the same properties.

```
F = fimath;

a = fi(pi);
b = fi(pi,F);
c = fi(pi/2,F);
d = fi(pi/2,0);

tf = ispropequal(a,b)
```

```
tf = logical
   1
```

```
tf = ispropequal(b,c)
```

```
tf = logical
   0
```

```
tf = ispropequal(c,d)
```

```
tf = logical
   0
```

Input Arguments

`a,b` — Inputs to be compared (as separate arguments)
arrays

Inputs to be compared, specified as arrays.

Data Types: `fi`

Tips

To compare the real-world values of two `fi` objects `a` and `b`, use `a == b` or `isequal(a,b)`.

Version History

Introduced before R2006a

See Also

`fi` | `isequal`

isquantizer

Determine whether input is `quantizer` object

Syntax

```
tf = isquantizer(q)
```

Description

`tf = isquantizer(q)` returns `1` (`true`) when `q` is a `quantizer` object. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Variable is a `quantizer` Object

Create a variable and determine whether it is a `quantizer` object.

```
q = quantizer('fixed', 'Ceiling', 'Wrap', [16 12])
```

```
q =


        DataMode = fixed
       RoundMode = ceil
    OverflowMode = wrap
          Format = [16  12]
```

```
tf = isquantizer(q)
```

```
tf = logical
   1
```

```
y = quantize(q,[pi pi/2])
```

```
y = 1×2

    3.1416    1.5708
```

```
tf = isquantizer(y)
```

```
tf = logical
   0
```

Input Arguments

q — Input array
array

Input array.

Version History

Introduced in R2008a

See Also

`quantizer` | `isfi` | `isfimath` | `isfipref` | `isnumerictype`

isscaleddouble

Determine whether input is scaled double data type

Syntax

```
tf = isscaleddouble(a)
tf = isscaleddouble(T)
```

Description

`tf = isscaleddouble(a)` returns `1` (`true`) when the `DataType` property of `fi` object `a` is `ScaledDouble`. Otherwise, it returns `0` (`false`).

`tf = isscaleddouble(T)` returns `1` (`true`) when the `DataType` property of `numerictype` object `T` is `ScaledDouble`. Otherwise, it returns `0` (`false`).

Examples

Determine Whether `fi` Object is a Scaled Double

Create a `fi` object and determine whether its `DataType` property is set to `ScaledDouble`.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isscaleddouble(a)
```

```
tf = logical
   0
```

```
T = numerictype('DataType','ScaledDouble');
a = fi(pi,T)
```

```
a = 
    3.1416


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isscaleddouble(a)
```

```
tf = logical
   1
```

Determine Whether numerictype Object is a Scaled Double

Create a `numerictype` object and determine whether its `DataType` property is set to `ScaledDouble`.

```
T = numerictype
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isscaleddouble(T)
```

```
tf = logical
   0
```

```
T = numerictype('DataType','ScaledDouble')
```

```
T =


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isscaleddouble(T)
```

```
tf = logical
   1
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2008a

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isscaledtype

Determine whether input is fixed-point or scaled double data type

Syntax

```
tf = isscaledtype(a)
tf = isscaledtype(T)
```

Description

`tf = isscaledtype(a)` returns `1` (`true`) when the `DataType` property of `fi` object `a` is `Fixed` or `ScaledDouble`. Otherwise, it returns `0` (`false`).

`tf = isscaledtype(T)` returns `1` (`true`) when the `DataType` property of `numerictype` object `T` is `Fixed` or `ScaledDouble`. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input is Fixed-Point or Scaled Double Data Type

Create a `fi` object and determine whether its `DataType` property is set to `Fixed` or `ScaledDouble`.

```
a = fi([pi,pi/2]);
tf = isscaledtype(a)
```

```
tf = logical
   1
```

Create a `numerictype` object and determine whether its `DataType` property is set to `Fixed` or `ScaledDouble`.

```
T1 = numerictype('DataType','ScaledDouble');
tf = isscaledtype(T1)
```

```
tf = logical
   1
```

```
T2 = numerictype('DataType','Single');
tf = isscaledtype(T2)
```

```
tf = logical
   0
```

Input Arguments

a — Input fi object
scalar | vector | matrix | multidimensional array

Input fi object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: fi

T — Input numerictype object
scalar

Input numerictype object, specified as a scalar.

Version History

Introduced in R2008a

See Also

isboolean | isdouble | isfixed | isfloat | numerictype | isscaleddouble | isscalingbinarypoint | isscalingslopebias | isscalingunspecified | issingle

isscalingbinarypoint

Determine whether input has binary point scaling

Syntax

```
tf = isscalingbinarypoint(a)
tf = isscalingbinarypoint(T)
```

Description

`tf = isscalingbinarypoint(a)` returns `1` (`true`) when the `fi` object `a` has binary point scaling or trivial slope and bias scaling. Otherwise, it returns `0` (`false`). Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

`tf = isscalingbinarypoint(T)` returns `1` (`true`) when the `numerictype` object `T` has binary point scaling or trivial slope and bias scaling. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input has Binary Point Scaling

Create a `fi` object and determine whether it has binary point scaling.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isscalingbinarypoint(a)
```

```
tf = logical
   1
```

```
b = fi(pi,1,16,3,2)
```

```
b = 
     2

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

```
tf = isscalingbinarypoint(b)
```

```
tf = logical
   0
```

If the `fi` object has trivial slope and bias scaling, that is, the slope is an integer power of two and the bias is zero, `isscalingbinarypoint` returns `1`.

```
c = fi(pi,1,16,4,0)
```

```
c = 
     4

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^2
                  Bias: 0
```

```
tf = isscalingbinarypoint(c)
```

```
tf = logical
   1
```

Create a `numerictype` object and determine whether it has binary point scaling.

```
T = numerictype
```

```
T = 


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isscalingbinarypoint(T)
```

```
tf = logical
   1
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2010b

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingslopebias` | `isscalingunspecified` | `issingle`

isscalingslopebias

Determine whether input has nontrivial slope and bias scaling

Syntax

```
tf = isscalingslopebias(a)
tf = isscalingslopebias(T)
```

Description

`tf = isscalingslopebias(a)` returns `1` (`true`) when the `fi` object `a` has nontrivial slope and bias scaling. Otherwise, it returns `0` (`false`). Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

`tf = isscalingslopebias(T)` returns `1` (`true`) when the `numerictype` object `T` has nontrivial slope and bias scaling. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input has Nontrivial Slope and Bias Scaling

Create a `fi` object and determine whether it has nontrivial slope and bias scaling.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isscalingslopebias(a)
```

```
tf = logical
   0
```

```
b = fi(pi,1,16,3,1)
```

```
b = 
     4

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 1
```

```
tf = isscalingslopebias(b)
```

```
tf = logical
   1
```

If the `fi` object has trivial slope and bias scaling, that is, the slope is an integer power of two and the bias is zero, `isscalingslopebias` returns `0`.

```
c = fi(pi,1,16,4,0)
```

```
c = 
     4

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 2^2
                  Bias: 0
```

```
tf = isscalingslopebias(c)
```

```
tf = logical
   0
```

Create a `numerictype` object and determine whether it has nontrivial slope and bias scaling.

```
T = numerictype
```

```
T = 


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isscalingslopebias(T)
```

```
tf = logical
   0
```

Input Arguments

a — Input `fi` object

scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object

scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2010b

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingunspecified` | `issingle`

isscalingunspecified

Determine whether input has unspecified scaling

Syntax

```
tf = isscalingunspecified(a)
tf = isscalingunspecified(T)
```

Description

`tf = isscalingunspecified(a)` returns `1` (`true`) if `fi` object `a` has a fixed-point or scaled double data type and its scaling has not been specified.

`tf = isscalingunspecified(T)` returns `1` (`true`) if `numerictype` object `T` has a fixed-point or scaled double data type and its scaling has not been specified.

Examples

Determine Whether Input has Unspecified Scaling

Create a `numerictype` object and determine whether it has unspecified scaling.

```
T1 = numerictype(0)
```

```
T1 =


          DataTypeMode: Fixed-point: unspecified scaling
            Signedness: Unsigned
            WordLength: 16
```

```
tf = isscalingunspecified(T1)
```

```
tf = logical
   1
```

```
T2 = numerictype(0,24,12,'DataType','ScaledDouble')
```

```
T2 =


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Unsigned
            WordLength: 24
        FractionLength: 12
```

```
tf = isscalingunspecified(T2)
```

```
tf = logical
   0
```

Create a `fi` object and determine whether it has unspecified scaling.

```
a = fi(pi,1)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = isscalingunspecified(a)
```

```
tf = logical
   0
```

Input Arguments

a — Input `fi` object

scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object

scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2010b

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `issingle`

issigned

Determine whether `fi` object is signed

Syntax

```
tf = issigned(a)
```

Description

`tf = issigned(a)` returns `1` (`true`) if the `fi` object `a` is signed. Otherwise, it returns `0` (`false`).

Examples

Determine Whether fi Object is Signed

Create a `fi` object and determine whether it is signed or unsigned.

```
a1 = fi(pi,1)
```

```
a1 = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = issigned(a1)
```

```
tf = logical
   1
```

```
a2 = fi(pi,0)
```

```
a2 = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 14
```

```
tf = issigned(a2)
```

```
tf = logical
   0
```

If a `numerictype` object with `Auto Signedness` is used to create a `fi` object, the `Signedness` property of the `fi` object automatically defaults to `Signed`.

```
T = numerictype('Signedness','Auto')
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Auto
            WordLength: 16
        FractionLength: 15
```

```
a3 = fi(pi,T)
```

```
a3 = 
    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = issigned(a3)
```

```
tf = logical
   1
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`isfi` | `isfixed` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified`

issingle

Determine whether input is single-precision data type

Syntax

```
tf = issingle(a)
tf = issingle(T)
```

Description

`tf = issingle(a)` returns `1` (`true`) when the `DataType` property of `fi` object `a` is `single`. Otherwise, it returns `0` (`false`).

`tf = issingle(T)` returns `1` (`true`) when the `DataType` property of `numerictype` object `T` is `single`. Otherwise, it returns `0` (`false`).

Examples

Determine Whether Input is Single-Precision Data Type

Create a `fi` object and determine whether it is single-precision data type.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
tf = issingle(a)
```

```
tf = logical
   0
```

Create a `numerictype` object and determine whether it is single-precision data type.

```
T = numerictype('Single')
```

```
T = 


          DataTypeMode: Single
```

```
tf = issingle(T)
```

```
tf = logical
   1
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2008a

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `isscalingbinarypoint` | `isscalingslopebias` | `isscalingunspecified`

isslopebiasscaled

Determine whether `numerictype` object has nontrivial slope and bias scaling

Syntax

```
tf = isslopebiasscaled(T)
```

Description

`tf = isslopebiasscaled(T)` returns `1` (`true`) when `numerictype` `T` has nontrivial slope and bias scaling. Otherwise, it returns `0` (`false`). Slope and bias scaling is trivial when the slope is an integer power of two and the bias is zero.

Examples

Determine Whether `numerictype` Object has Nontrivial Slope and Bias Scaling

Create a `numerictype` object and determine whether it has nontrivial slope and bias scaling.

```
T1 = numerictype
```

```
T1 =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
tf = isslopebiasscaled(T1)
```

```
tf = logical
   0
```

```
T2 = numerictype('DataTypeMode',...
    'Fixed-point: slope and bias scaling',...
    'WordLength',32,...
    'Slope',2^-2,...
    'Bias',4)
```

```
T2 =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 32
                 Slope: 0.25
                  Bias: 4
```

```
tf = isslopebiasscaled(T2)
```

```
tf = logical
   1

T3 = numerictype('DataTypeMode',...
    'Fixed-point: slope and bias scaling',...
    'WordLength',32,...
    'Slope',2^2,...
    'Bias',0)

T3 =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 32
                 Slope: 2^2
                  Bias: 0

tf = isslopebiasscaled(T3)

tf = logical
   0
```

Input Arguments

T — Input `numerictype` object
scalar

Input `numerictype` object, specified as a scalar.

Version History

Introduced in R2008a

See Also

`isboolean` | `isdouble` | `isfixed` | `isfloat` | `isscaleddouble` | `isscaledtype` | `issingle` | `numerictype`

le, <=

Package: `embedded`

Determine whether real-world value of one array is less than or equal to another

Syntax

```
A <= B
le(A,B)
```

Description

`A <= B` returns a logical array with elements set to logical `1` (`true`) where the real-world values of `A` is less than or equal to `B`, when `A` or `B` is a `fi` object. Otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to a fixed-point type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object.

`le(A,B)` is an alternate way to execute `A <= B`, but is rarely used.

Examples

Compare Two `fi` Objects

Use the `le` function to determine whether the real-world value of one `fi` object is less than or equal to another.

```
a = fi(pi);
b = fi(pi, 1, 32);
a <= b
```

```
ans = logical
   0
```

Input `a` has a 16-bit word length, while input `b` has a 32-bit word length. The `le` function returns `0` because after quantization, the value of `a` is greater than that of `b`.

Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the floating-point double is cast to a type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

```
a = fi(pi);
b = pi;
le(a,b)

ans =

  logical

   0
```

Input Arguments

A,B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Version History

Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators
Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for fi le, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a try/catch block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

R2022a: Improved accuracy in comparing fi objects and floating-point numbers using relational operators
Behavior changed in R2022a

In previous releases, when comparing a single or double to a fi object, the floating-point value was cast to the same word length and signedness of the fi object. This could lead to incorrect results. For example,

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   0   0
```

```
fi(65534)
fi(65534.25) == 65534.25

ans = 

       65534

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: -1

ans =

  logical

   1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   1   0
```

Note that the updated algorithm may produce subtle, but accurate, results. For example:

```
fi(pi) == pi

ans =

  logical

   0
```

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest wordlength growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eq` | `ge` | `gt` | `lt` | `ne`

logreport

Quantization report

Syntax

```
logreport(a)
logreport(a, b, ...)
```

Description

`logreport(a)` displays the `minlog`, `maxlog`, `lowerbound`, `upperbound`, `noverflows`, and `nunderflows` for the `fi` object `a`.

`logreport(a, b, ...)` displays the report for each `fi` object `a, b, ...`.

Examples

The following example produces a `logreport` for `fi` objects `a` and `b`:

```
fipref('LoggingMode','On');
a = fi(pi);
b = fi(randn(10),1,8,7);
```

```
Warning: 35 overflow(s) occurred in the fi assignment operation.
> In embedded.fi/fifactory
In fi (line 226)
Warning: 2 underflow(s) occurred in the fi assignment operation.
> In embedded.fi/fifactory
In fi (line 226)
```

```
logreport(a,b)
```

```
logreport(a,b)
              minlog       maxlog   lowerbound   upperbound   noverflows  nunderflows
       a    3.141602     3.141602           -4     3.999878            0            0
       b          -1    0.9921875           -1    0.9921875           35            2
```

Version History

Introduced in R2008a

See Also

`fipref` | `quantize` | `quantizer`

lowerbound

Package: `embedded`

Lower bound of range of `fi` object

Syntax

`l = lowerbound(a)`

Description

`l = lowerbound(a)` returns the lower bound of the range of `fi` object `a`.

If `l = lowerbound(a)` and `u = upperbound(a)`, then `[l,u] = range(a)`.

Examples

Lower Bound of `fi` Object

```
a = fi(pi,1,16,3,2)
```

```
a = 
     2

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

```
l = lowerbound(a)
```

```
l = 
      -98302

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

Input Arguments

a — Input `fi` object
`fi` object

Input `fi` object.

Data Types: `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eps` | `intmax` | `intmin` | `lsb` | `range` | `realmax` | `realmin` | `upperbound`

lsb

Package: `embedded`

Scaling of least significant bit of `fi` object, or value of least significant bit of `quantizer` object

Syntax

```
b = lsb(a)
p = lsb(q)
```

Description

`b = lsb(a)` returns the scaling of the least significant bit of `fi` object `a`. The result is equivalent to the result given by the `eps` function.

`p = lsb(q)` returns the quantization level of `quantizer` object `q` or the distance from `1.0` to the next largest floating-point number if `q` is a floating-point `quantizer` object.

Examples

Least Significant Bit of `fi` Object

Use the `lsb` function to find the value of the scaling of the least significant bit of `fi` object `a`.

Create a signed fi object that specifies a word length of 8 bits and a fraction length of 7 bits.

```
a = fi([],1,8,7)
```

```
a =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Determine the least significant bit of the `fi` object.

```
lsb(a)
```

```
ans =
    0.0078

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Least Significant Bit of quantizer Object

Use the `lsb` function to find the value of the least significant bit of the `quantizer` object `q`.

Create a fixed-point `quantizer` object that specifies a word length of 8 bits and a fraction length of 7 bits.

```
q = quantizer('fixed',[8 7])
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [8  7]
```

Determine the quantization level of the `quantizer` object.

```
p = lsb(q)
```

```
p = 0.0078
```

For both fixed-point and floating-point `quantizer` objects `q`, `lsb(q) = 2^-FRACTIONLENGTH(q)`.

```
lsb(q)==2^-7
```

```
ans = logical
   1
```

Input Arguments

a — Input array
`fi` object

Input array, specified as a `fi` object.

Data Types: `fi`
Complex Number Support: Yes

q — Input `quantizer` object
`quantizer` object

Input `quantizer` object, specified as a `quantizer` object.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation supports scalar fixed-point signals only.
- Code generation supports scalar, vector, and matrix, `fi` single and `fi` double signals.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eps` | `intmax` | `intmin` | `lowerbound` | `quantize` | `range` | `realmax` | `realmin` | `upperbound`

lt, <

Package: `embedded`

Determine whether real-world value of one array is less than another

Syntax

```
A < B
lt(A,B)
```

Description

`A < B` returns a logical array with elements set to logical `1` (`true`) where the real-world values of `A` is less than `B`, when `A` or `B` is a `fi` object. Otherwise, the element is logical `0` (`false`). The test compares only the real part of numeric arrays.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to a fixed-point type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object.

`lt(A,B)` is an alternate way to execute `A < B`, but is rarely used.

Examples

Compare Two `fi` Objects

Use the `lt` function to determine whether the real-world value of one `fi` object is less than another.

```
a = fi(pi);
b = fi(pi, 1, 32);
a < b
```

```
ans = logical
   0
```

Input `a` has a 16-bit word length, while input `b` has a 32-bit word length. The `lt` function returns `0` because after quantization, the value of `a` is greater than that of `b`.

Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the floating-point double is cast to a type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

```
a = fi(pi);
b = pi;
lt(a,b)

ans =

  logical

   0
```

Input Arguments

A,B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see "Compatible Array Sizes for Basic Operations".

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Version History
Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators
Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for fi lt, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a try/catch block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see "Compatible Array Sizes for Basic Operations".

R2022a: Improved accuracy in comparing fi objects and floating-point numbers using relational operators
Behavior changed in R2022a

In previous releases, when comparing a single or double to a fi object, the floating-point value was cast to the same word length and signedness of the fi object. This could lead to incorrect results. For example,

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   0   0
```

```
fi(65534)
fi(65534.25) == 65534.25

ans = 

       65534

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: -1

ans =

  logical

   1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   1   0
```

Note that the updated algorithm may produce subtle, but accurate, results. For example:

```
fi(pi) == pi

ans =

  logical

   0
```

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest wordlength growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eq` | `ge` | `gt` | `le` | `ne`

mat2str

Convert matrix to string

Syntax

```
str = mat2str(A)
str = mat2str(A, n)
str = mat2str(A, 'class')
str = mat2str(A, n, 'class')
```

Description

`str = mat2str(A)` converts `fi` object `A` to a string representation. The output is suitable for input to the `eval` function such that `eval(str)` produces the original `fi` object exactly.

`str = mat2str(A, n)` converts `fi` object `A` to a string representation using `n` bits of precision.

`str = mat2str(A, 'class')` creates a string representation with the name of the class of `A` included. This option ensures that the result of evaluating `str` will also contain the class information.

`str = mat2str(A, n, 'class')` uses `n` bits of precision and includes the class of `A`.

Examples

Convert fi Object to a String

Convert the `fi` object `a` to a string.

```
a = fi(pi);
str = mat2str(a)

str =
'3.1416015625'
```

Convert fi Object to a String with Specified Precision

Convert the `fi` object `a` to a string using eight bits of precision.

```
a = fi(pi);
str = mat2str(a, 8)

str =
'3.1416016'
```

Input Arguments

A — Input array
scalar | vector | matrix

Input array, specified as a scalar, vector, or matrix. A cannot be a multidimensional array.

Data Types: fi|single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

n — Number of bits of precision
positive integer

Number of bits of precision in the output, specified as a positive integer.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

str — String representation of input array
character array

String representation of input array, returned as a character array.

Version History

Introduced in R2015b

See Also

mat2str | tostring

max

Largest element in array of `fi` objects

Syntax

```
M = max(A)
M = max(A,[],dim)
[M,I] = max( ___ )

C = max(A,B)
```

Description

`M = max(A)` returns the largest elements along different dimensions of `fi` array `A`.

- If `A` is a vector, `max(A)` returns the largest element in `A`.
- If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column.
- If `A` is a multidimensional array, `max` operates along the first nonsingleton dimension and returns an array of maximum values.

`M = max(A,[],dim)` returns the largest elements along dimension `dim`.

`[M,I] = max( ___ )` finds the indices of the maximum values and returns them in array `I`, using any of the input arguments in the previous syntaxes. If the largest value occurs multiple times, the index of the first occurrence is returned.

`C = max(A,B)` returns an array with the largest elements taken from `A` or `B`.

Examples

Largest Element in a Vector

Create a fixed-point vector and return the maximum value from the vector.

```
A = fi([1,5,4,9,2],1,16);
M = max(A)
```

```
M = 
     9

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Largest Element of Each Matrix Row

Create a fixed-point matrix.

```
A = fi(magic(4),1,16)
```

```
A = 
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

Find the largest element of each row by finding the maximum values along the second dimension.

```
M = max(A,[],2)
```

```
M = 
    16
    11
    12
    15

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

The output vector, M, is a column vector that contains the largest element of each row.

Largest Element of Each Matrix Column

Create a fixed-point matrix.

```
A = fi(magic(4),1,16)
```

```
A = 
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

Find the largest element of each column.

```
M = max(A)
```

```
M = 
    16    14    15    13
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

The output, `M`, is a row vector that contains the largest elements from each column of `A`.

Find the index of each of the maximum elements.

```
[M,I] = max(A)
```

```
M = 
    16    14    15    13

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10

I = 1×4

     1     4     4     1
```

Vector `I` contains the indices to the minimum elements in `M`.

Maximum Elements from Two Arrays

Create two fixed-point arrays of the same size.

```
A = fi([2.3,4.7,6;0,7,9.23],1,16);
B = fi([9.8,3.21,1.6;pi,2.3,1],1,16);
```

Find the largest elements from `A` or `B`.

```
C = max(A,B)
```

```
C = 
    9.7998    4.7002    6.0000
    3.1416    7.0000    9.2300

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

`C` contains the largest elements from each pair of corresponding elements in `A` and `B`.

Largest Element of a Complex Vector

Create a complex fixed-point vector, `a`.

```
a = fi([1+2i,3+6i,6+3i,2-4i],1,16)
```

```
a =
   1.0000 + 2.0000i   3.0000 + 6.0000i   6.0000 + 3.0000i   2.0000 - 4.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

The function finds the largest element of a complex vector by taking the element with the largest magnitude.

```
abs(a)
```

```
ans =
    2.2361    6.7083    6.7083    4.4722

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

In vector `a`, the largest elements, at position `2` and `3`, have a magnitude of `6.7083`. The `max` function returns the largest element in output `x` and the index of that element in output `y`.

```
[x,y] = max(a)
```

```
x =
   3.0000 + 6.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12

y = 2
```

Although the elements at index 2 and 3 have the same magnitude, the index of the first occurrence of that value is always returned.

Input Arguments

A — Input `fi` array
scalar | vector | matrix | multidimensional array

Input `fi` array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of `A` and `B` must match unless one is a scalar.

The `max` function ignores `NaNs`.

Data Types: `fi`

Complex Number Support: Yes

B — Additional input array
scalar | vector | matrix | multidimensional array

Additional input `fi` or numeric array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of `A` and `B` must match unless one is a scalar.

The `max` function ignores `NaNs`.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

`dim` — dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. `dim` can also be a `fi` object. If you do not specify a value, the default value is the first array dimension whose size does not equal 1.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`M` — Maximum values
scalar | vector | matrix | multidimensional array

Maximum values, returned as a scalar, vector, matrix, or multidimensional array. `M` always has the same data type as the input.

`I` — Index
scalar | vector | matrix | multidimensional array

Index, returned as a scalar, vector, matrix, or multidimensional array. If the largest value occurs more than once, then `I` contains the index to the first occurrence of the value. `I` is always of data type `double`.

`C` — Maximum elements from `A` or `B`
scalar | vector | matrix | multidimensional array

Maximum elements from `A` or `B`, returned as a scalar, vector, matrix, or multidimensional array.

Algorithms

When `A` or `B` is complex, the `max` function returns the elements with the largest magnitude. If two magnitudes are equal, then `max` returns the first value. This behavior differs from how the built-in `max` function resolves ties between complex numbers.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`mean` | `median` | `min` | `sort`

maxlog

Log maximums

Syntax

```
y = maxlog(a)
y = maxlog(q)
```

Description

`y = maxlog(a)` returns the largest real-world value of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` object `LoggingMode` property to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = maxlog(q)` is the maximum value after quantization during a call to `quantize(q,...)` for `quantizer` object `q`. This value is the maximum value encountered over successive calls to `quantize` since logging was turned on, and is reset with `resetlog(q)`. `maxlog(q)` is equivalent to `get(q,'maxlog')` and `q.maxlog`.

Examples

Example 1: Using maxlog with fi objects

1

```
P = fipref('LoggingMode','on');
format long g
a = fi([-1.5 eps 0.5], true, 16, 15);
a(1) = 3.0;
maxlog(a)
```

```
Warning: 1 overflow(s) occurred in the fi assignment operation.
> In embedded.fi/fifactory
In fi (line 226)
Warning: 1 underflow(s) occurred in the fi assignment operation.
> In embedded.fi/fifactory
In fi (line 226)
Warning: 1 overflow(s) occurred in the fi assignment operation.

ans =

         0.999969482421875
```

The largest value `maxlog` can return is the maximum representable value of its input. In this example, `a` is a signed `fi` object with word length `16`, fraction length `15` and range:

$$-1 \le x \le 1 - 2^{-15} \quad (4\text{-}10)$$

2 You can obtain the numerical range of any `fi` object `a` using the `range` function:

```
format long g
r = range(a)

r =

                        -1         0.999969482421875

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Example 2: Using `maxlog` with `quantizer` objects

1
```
q = quantizer;
warning on
format long g
x = [-20:10];
y = quantize(q,x);
maxlog(q)

Warning: 29 overflow(s) occurred in the fi quantize operation.
> In embedded.quantizer/quantize (line 81)

ans =

         0.999969482421875
```

The largest value `maxlog` can return is the maximum representable value of its input.

2 You can obtain the range of `x` after quantization using the `range` function:

```
format long g
r = range(q)

r =

                        -1         0.999969482421875
```

Version History

Introduced before R2006a

See Also

`fipref` | `minlog` | `noverflows` | `nunderflows` | `reset` | `resetlog`

mean

Average or mean value of fixed-point array

Syntax

```
M = mean(A)
M = mean(A,dim)
```

Description

`M = mean(A)` computes the mean value of the real-valued fixed-point array `A` along its first nonsingleton dimension.

`M = mean(A,dim)` computes the mean value of the real-valued fixed-point array `A` along dimension `dim`. `dim` must be a positive, real-valued integer with a power-of-two slope and a bias of 0.

The fixed-point output array, `M`, has the same `numerictype` properties as the fixed-point input array, `A`.

If the input array, `A`, has a local `fimath`, then it is used for intermediate calculations. The output, `M`, is always associated with the default `fimath`.

When `A` is an empty fixed-point array (value = `[]`), the value of the output array is zero.

Examples

Mean Along Columns of Fixed-Point Array

Create a matrix and compute the mean of each column. `A` is a signed `fi` object with a 32-bit word length and a best-precision fraction length of 28 bits.

```
A = fi([0 1 2; 3 4 5],1,32);
M = mean(A)
```

```
A = 

     0     1     2
     3     4     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28

M = 

    1.5000    2.5000    3.5000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
          WordLength: 32
      FractionLength: 28
```

Mean Along Rows of Fixed-Point Array

Create a matrix and compute the mean of each row. `A` is a signed `fi` object with a 32-bit word length and a best-precision fraction length of 28 bits.

```
A = fi([0 1 2; 3 4 5],1,32)
M = mean(A,2)
```

```
A = 

     0     1     2
     3     4     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28

M = 

     1
     4

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

- If `A` is a scalar, then `mean(A)` returns `A`.
- If `A` is an empty fixed-point array (value = `[]`), the value of the output array is zero.

Data Types: `fi`

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive, real-valued, integer scalar with a power-of-two slope and a bias of 0. If no value is specified, then the default is the first array dimension whose size does not equal 1.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Algorithms

The general equation for computing the `mean` of an array `A`, across dimension `dim` is:

```
sum(A,dim)/size(A,dim)
```

Because `size(a,dim)` is always a positive integer, the algorithm for computing mean casts `size(A,dim)` to an unsigned 32-bit `fi` object with a fraction length of zero (denote this `fi` object `'SizeA'`). The algorithm then computes the mean of `A` according to the following equation, where `Tx` represents the `numerictype` properties of the fixed-point input array `A`:

```
c = Tx.divide(sum(A,dim), SizeA)
```

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`max` | `median` | `min`

median

Median value of fixed-point array

Syntax

c = median(*a*)
c = median(*a*,*dim*)

Description

c = median(*a*) computes the median value of the fixed-point array *a* along its first nonsingleton dimension.

c = median(*a*,*dim*) computes the median value of the fixed-point array *a* along dimension *dim*. *dim* must be a positive, real-valued integer with a power-of-two slope and a bias of 0.

The input to the median function must be a real-valued fixed-point array.

The fixed-point output array *c* has the same numerictype properties as the fixed-point input array *a*. If the input, *a*, has a local fimath, then it is used for intermediate calculations. The output, *c*, is always associated with the default fimath.

When *a* is an empty fixed-point array (value = []), the value of the output array is zero.

Examples

Compute the median value along the first dimension of a fixed-point array.

```
x = fi([0 1 2; 3 4 5; 7 2 2; 6 4 9], 1, 32)
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 27 bits
mx1 = median(x,1)
```

Compute the median value along the second dimension (columns) of a fixed-point array.

```
x = fi([0 1 2; 3 4 5; 7 2 2; 6 4 9], 1, 32)
% x is a signed FI object with a 32-bit word length
% and a best-precision fraction length of 27 bits
mx2 = median(x, 2)
```

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

max | mean | min

min

Smallest element in array of `fi` objects

Syntax

```
M = min(A)
M = min(A,[],dim)
[M,I] = min( ___ )

C = min(A,B)
```

Description

`M = min(A)` returns the smallest elements along different dimensions of `fi` array `A`.

- If `A` is a vector, `min(A)` returns the smallest element in `A`.
- If `A` is a matrix, `min(A)` treats the columns of `A` as vectors, returning a row vector containing the minimum element from each column.
- If `A` is a multidimensional array, `min` operates along the first nonsingleton dimension and returns an array of minimum values.

`M = min(A,[],dim)` returns the smallest elements along dimension `dim`.

`[M,I] = min( ___ )` finds the indices of the minimum values and returns them in array `I`, using any of the input arguments in the previous syntaxes. If the smallest value occurs multiple times, the index of the first occurrence is returned.

`C = min(A,B)` returns an array with the smallest elements taken from `A` or `B`.

Examples

Smallest Element in a Vector

Create a fixed-point vector and return the minimum value from the vector.

```
A = fi([1,5,4,9,2],1,16);
M = min(A)
```

```
M = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Minimum Element of Each Matrix Row

Create a matrix of fixed-point values.

```
A = fi(magic(4),1,16)
```

```
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

Find the smallest element of each row by finding the minimum values along the second dimension.

```
M = min(A,[],2)
```

```
M =
     2
     5
     6
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

The output, M, is a column vector that contains the smallest element of each row of A.

Minimum Element of Each Matrix Column

Create a fixed-point matrix.

```
A = fi(magic(4),1,16)
```

```
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

Find the smallest element of each column.

```
M = min(A)
```

```
M =
     4     2     3     1
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

The output, M, is a row vector that contains the smallest element of each column of A.

Find the index of each of the minimum elements.

```
[M,I] = min(A)
```

```
M = 
     4     2     3     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10

I = 1×4

     4     1     1     4
```

Minimum Elements from Two Arrays

Create two fixed-point arrays of the same size.

```
A = fi([2.3,4.7,6;0,7,9.23],1,16);
B = fi([9.8,3.21,1.6;pi,2.3,1],1,16);
```

Find the minimum elements from A or B.

```
C = min(A,B)
```

```
C = 
    2.2998    3.2100    1.6001
         0    2.2998    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

C contains the smallest elements from each pair of corresponding elements in A and B.

Minimum Element of a Complex Vector

Create a complex fixed-point vector, A.

```
A = fi([1+2i,2+i,3+8i,9+i],1,8)
```

```
A =
   1.0000 + 2.0000i   2.0000 + 1.0000i   3.0000 + 8.0000i   9.0000 + 1.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

The `min` function finds the smallest element of a complex vector by taking the element with the smallest magnitude.

```
abs(A)
```

```
ans =
    2.2500    2.2500    8.5000    9.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

In vector A, the smallest elements, at position `1` and `2`, have a magnitude of `2.25`. The `min` function returns the smallest element in output M, and the index of that element in output, I.

```
[M,I] = min(A)
```

```
M =
   1.0000 + 2.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3

I = 1
```

Although the elements at index 1 and 2 have the same magnitude, the index of the first occurrence of that value is always returned.

Input Arguments

A — Input `fi` array

scalar | vector | matrix | multidimensional array

`fi` or numeric input array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of `A` and `B` must match unless one is a scalar.

The `min` function ignores `NaN`s.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

B — Additional input array

scalar | vector | matrix | multidimensional array

Additional input `fi` or numeric array, specified as a scalar, vector, matrix, or multidimensional array. The dimensions of `A` and `B` must match unless one is a scalar.

The `min` function ignores `NaN`s.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Complex Number Support: Yes

`dim` — dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. `dim` can also be a `fi` object. If you do not specify a value, the default value is the first array dimension whose size does not equal 1.

Data Types: `fi`|`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`M` — Minimum values
scalar | vector | matrix | multidimensional array

Minimum values, returned as a scalar, vector, matrix, or multidimensional array. `M` always has the same data type as the input.

`I` — Index
scalar | vector | matrix | multidimensional array

Index, returned as a scalar, vector, matrix, or multidimensional array. If the smallest value occurs more than once, then `I` contains the index to the first occurrence of the value. `I` is always of data type `double`.

`C` — Minimum elements from `A` or `B`
scalar | vector | matrix | multidimensional array

Minimum elements from `A` or `B`, returned as a scalar, vector, matrix, or multidimensional array.

Algorithms

When `A` or `B` is complex, the `min` function returns the element with the smallest magnitude. If two magnitudes are equal, then `min` returns the first value. This behavior differs from how the built-in `min` function resolves ties between complex numbers.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`mean` | `median` | `max` | `sort`

minlog

Log minimums

Syntax

```
y = minlog(a)
y = minlog(q)
```

Description

`y = minlog(a)` returns the smallest real-world value of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` object `LoggingMode` property to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = minlog(q)` is the minimum value after quantization during a call to `quantize(q,...)` for `quantizer` object `q`. This value is the minimum value encountered over successive calls to `quantize` since logging was turned on, and is reset with `resetlog(q)`. `minlog(q)` is equivalent to `get(q,'minlog')` and `q.minlog`.

Examples

Example 1: Using `minlog` with `fi` objects

1.
```
P = fipref('LoggingMode','on');
a = fi([-1.5 eps 0.5], true, 16, 15);
a(1) = 3.0;
minlog(a)
```

```
Warning: 1 overflow(s) occurred in the fi assignment operation.
> In embedded.fi/fifactory
In fi (line 226)
Warning: 1 underflow(s) occurred in the fi assignment operation.
> In embedded.fi/fifactory
In fi (line 226)
Warning: 1 overflow(s) occurred in the fi assignment operation.

ans =

    -1
```

The smallest value `minlog` can return is the minimum representable value of its input. In this example, `a` is a signed `fi` object with word length `16`, fraction length `15` and range:

$$-1 \leq x \leq 1 - 2^{-15} \quad (4\text{-}11)$$

2. You can obtain the numerical range of any `fi` object `a` using the `range` function:

```
format long g
r = range(a)
```

```
r =

                       -1         0.999969482421875

           DataTypeMode: Fixed-point: binary point scaling
             Signedness: Signed
             WordLength: 16
         FractionLength: 15
```

Example 2: Using `minlog` with `quantizer` objects

1

```
q = quantizer;
warning on
x = [-20:10];
y = quantize(q,x);
minlog(q)
```

```
Warning: 29 overflow(s) occurred in the fi quantize operation.
> In embedded.quantizer/quantize (line 81)

ans =

    -1
```

The smallest value `minlog` can return is the minimum representable value of its input.

2 You can obtain the range of `x` after quantization using the `range` function:

```
format long g
r = range(q)

r =

                       -1         0.999969482421875
```

Version History

Introduced before R2006a

See Also

`fipref` | `maxlog` | `noverflows` | `nunderflows` | `reset` | `resetlog`

minus, -

Package: embedded

Matrix difference between fi objects

Syntax

```
C = A-B
C = minus(A,B)
```

Description

C = A-B subtracts matrix B from matrix A.

minus does not support fi objects of data type boolean.

C = minus(A,B) is an alternate way to execute A-B.

Note For information about the fimath properties involved in Fixed-Point Designer calculations, see "fimath Properties Usage for Fixed-Point Arithmetic" and "fimath ProductMode and SumMode".

Input Arguments

A — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fi objects or built-in data types. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see "Compatible Array Sizes for Basic Operations".

minus does not support fi objects of data type boolean.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

B — Input array

scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fi objects or built-in data types. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see "Compatible Array Sizes for Basic Operations".

minus does not support fi objects of data type boolean.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Version History

Introduced before R2006a

R2021b: Implicit expansion change affects arguments for operators

Behavior changed in R2021b

Starting in R2021b with the addition of implicit expansion for `fi` `times`, `plus`, and `minus`, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`mtimes` | `plus` | `times` | `uminus`

mod

Modulus after division for `fi` objects

Syntax

```
m = mod(x,y)
```

Description

`m = mod(x,y)` returns the modulus after division of `x` by `y`, where `x` is the dividend and `y` is the divisor. This function is often called the modulo operation, which can be expressed as `m = x - floor(x./y).*y`.

For fixed-point or integer input arguments, the output data type is the aggregate type of both input signedness, word lengths, and fraction lengths. For floating-point input arguments, the output data type is the same as the inputs.

The `mod` function ignores and discards any `fimath` attached to the inputs. The output is always associated with the default `fimath`.

Note The combination of fixed-point and floating-point inputs is not supported.

Examples

Modulus of two `fi` Objects

Calculate the `mod` of two `fi` objects.

```
x = fi(-3,1,7,0);
y = fi(2,1,15,0);
m1 = mod(x,y)
m2 = mod(y,x)
```

```
m1 =

     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 15
        FractionLength: 0

m2 =

    -1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
          WordLength: 15
      FractionLength: 0
```

Modulus of Two Floating-Point Inputs

Convert the `fi` inputs in the previous example to double type and calculate the `mod`.

```
Mf1 = mod(double(x),double(y))
Mf2 = mod(double(y),double(x))
```

```
Mf1 =

     1


Mf2 =

    -1
```

Input Arguments

x — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a scalar, vector, matrix, or multidimensional array. `x` must be a real-valued integer, fixed-point, or floating-point array, or real scalar. Numeric inputs `x` and `y` must either be the same size, or have sizes that are compatible.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

y — Divisor
scalar | vector | matrix | multidimensional array

Divisor, specified as a scalar, vector, matrix, or multidimensional array. `y` must be a real-valued integer, fixed-point, or floating-point array, or real scalar. Numeric inputs `x` and `y` must either be the same size, or have sizes that are compatible.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

m — Result of modulus operation
scalar | vector | matrix | multidimensional array

Result of modulus operation, returned as a scalar, vector, matrix, or multidimensional array.

If both inputs `x` and `y` are floating-point, then the data type of `m` is the same as the inputs. If either input `x` or `y` is fixed-point, then the data type of `m` is the aggregate `numerictype`. This value equals that of `fixed.aggregateType(x,y)`.

The output `m` is always associated with the default `fimath`.

Algorithms

`mod(x,y)` for a `fi` object uses the same definition as the built-in MATLAB `mod` function.

Version History

Introduced in R2011b

See Also

`fixed.aggregateType` | `mod`

modByConstant

Modulus after division by a constant denominator

Syntax

`Y = modByConstant(X,d)`

Description

`Y = modByConstant(X,d)` performs the modulo operation (remainder after division) of `X` with respect to the denominator `d`.

For simulation, the data type of the output `Y` is chosen based on the value of the denominator `d` and the range of `X`.

To generate code, the denominator `d` must be a constant.

Examples

Modulo by Constant Denominator

```
modByConstant(fi(10203),10)
```

```
ans = 

     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 5
        FractionLength: 1
```

```
modByConstant(uint16(6930),1024)
```

```
ans = 

   786

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 10
        FractionLength: 0
```

Input Arguments

X — Dividend
scalar | vector | matrix | multidimensional array

Dividend, specified as a scalar, vector, matrix, or multidimensional array.

If `X` is a fixed-point or scaled-double `fi`, it must use binary point scaling.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

d — Divisor
positive scalar

Divisor, specified as a positive, real-valued scalar.

If `d` is a fixed-point or scaled-double `fi`, it must use binary point scaling.

To generate code, the denominator `d` must be a constant.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Output Arguments

Y — Result of modulus operation
scalar | vector | matrix | multidimensional array

Result of modulus operation, returned as a scalar, vector, matrix, or multidimensional array.

For simulation, the data type of the output `Y` is chosen based on the value of the denominator `d` and the range of `X`.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

mpower, ^

Package: embedded

Fixed-point matrix power (^)

Syntax

```
Y = A^k
Y = mpower(A,k)
```

Description

Y = A^k computes A to the k power for fi inputs and returns the result in Y.

The matrix power operation is performed using default fimath settings.

The fixed-point output array Y has the same local fimath as the input A. If A has no local fimath, the output Y also has no local fimath.

Y = mpower(A,k) is an alternate way to execute A^k.

Examples

Square a Matrix

Compute the power of a 2-dimensional square matrix for exponent values 0, 1, 2, and 3.

```
x = fi([0 1; 2 4], 1, 32);
px0 = x^0
```

```
px0 = 
     1     0
     0     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 1
        FractionLength: 0
```

```
px1 = x^1
```

```
px1 = 
     0     1
     2     4

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 28
```

```
px2 = x^2
```

```
px2 =
     2     4
     8    18

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 65
        FractionLength: 56
```

```
px3 = x^3
```

```
px3 =
     8    18
    36    80

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 98
        FractionLength: 84
```

Input Arguments

A — Base
scalar | matrix

Base, specified as a scalar or matrix.

Example: `x = fi([0 1; 2 4],1,32);`

Data Types: `fi`
Complex Number Support: Yes

k — Exponent
positive real-valued integer

Exponent, specified as a real-valued integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Version History

Introduced in R2010a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the exponent `k` is a variable and the input is a scalar, the `ProductMode` property of the governing `fimath` must be `SpecifyPrecision`.
- When the exponent `k` is a variable and the input is not scalar, the `SumMode` property of the governing `fimath` must be `SpecifyPrecision`.

- Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` is set to `SpecifyPrecision` or `Keep LSB`.
- For variable-sized signals, you may see different results between the generated code and MATLAB.
  - In the generated code, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath`.
  - In MATLAB, the output for variable-sized signals is computed using the `SumMode` property of the governing `fimath` when the first input, `A`, is nonscalar. However, when `A` is a scalar, MATLAB computes the output using the `ProductMode` of the governing `fimath`.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Both inputs must be scalar, and the exponent input, `k`, must be a constant integer.

See Also
`mpower` | `power` | `fi` | `fimath`

mpy

Multiply two objects using `fimath` object

Syntax

`c = mpy(F,a,b)`

Description

`c = mpy(F,a,b)` performs elementwise multiplication on `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output `fi` object `c` has no local fimath.

`a` and `b` can both be `fi` objects with the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object. `a` and `b` can also be doubles, singles, or integers.

Examples

In this example, `c` is the 40-bit product of `a` and `b` with fraction length 30.

```
a = fi(pi);
b = fi(exp(1));
F = fimath('ProductMode','SpecifyPrecision',...
        'ProductWordLength',40,'ProductFractionLength',30);
c = F.mpy(a, b)
```

```
c =

    8.5397

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 30
```

Algorithms

`c = mpy(F,a,b)` is similar to

```
a.fimath = F;
b.fimath = F;
c = a .* b
```

```
c =

    8.5397

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
```

```
        FractionLength: 30

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: SpecifyPrecision
     ProductWordLength: 40
 ProductFractionLength: 30
               SumMode: FullPrecision
```

but not identical. When you use `mpy`, the `fimath` properties of `a` and `b` are not modified, and the output `fi` object `c` has no local fimath. When you use the syntax `c = a .* b`, where `a` and `b` have their own `fimath` objects, the output `fi` object `c` gets assigned the same `fimath` object as inputs `a` and `b`. See "fimath Rules for Fixed-Point Arithmetic" in the Fixed-Point Designer User's Guide for more information.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the syntax `F.mpy(a,b)`. You must use the syntax `mpy(F,a,b)`.
- When you provide complex inputs to the `mpy` function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Model Explorer and set the **Complexity** parameter for all known complex inputs to `On`.

See Also

`add` | `divide` | `fi` | `fimath` | `mrdivide` | `numerictype` | `rdivide` | `sub` | `sum`

mrdivide, /

Package: `embedded`

Right-matrix division

Syntax

```
X = A/b
X = mrdivide(A, b)
```

Description

`X = A/b` performs right-matrix division.

`X = mrdivide(A, b)` is an alternative way to execute `X = A/b`.

Examples

Divide fi Matrix by a Constant

In this example, you use the forward slash (/) operator to perform right matrix division on a 3-by-3 magic square of `fi` objects. Because the numerator input is a `fi` object, the denominator input `b` must be a scalar.

```
A = fi(magic(3))
```

```
A =
     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
b = fi(3,1,12,8)
```

```
b =
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 12
        FractionLength: 8
```

```
X = A/b
```

```
X =
    2.6250    0.3750    2.0000
    1.0000    1.6250    2.3750
    1.3750    3.0000    0.6250
```

```
      DataTypeMode: Fixed-point: binary point scaling
        Signedness: Signed
        WordLength: 16
    FractionLength: 3
```

Perform Matrix Division

You can perform right-matrix division when neither input is a `fi` object. The matrix dimensions must be compatible for matrix division.

```
A = [2, 3, 1; 0, 8, 4; 1, 1, 0]
```

```
A = 3×3

     2     3     1
     0     8     4
     1     1     0
```

```
B = [7, 6, 6; 1, 0, 5; 9, 0, 4]
```

```
B = 3×3

     7     6     6
     1     0     5
     9     0     4
```

```
X = mrdivide(A,B)
```

```
X = 3×3

    0.5000   -0.2927   -0.1341
    1.3333    0.0325   -1.0407
    0.1667   -0.2033    0.0041
```

Input Arguments

A — Numerator
scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array. If one or both of the inputs is a `fi` object, then `b` must be a scalar. When `b` is a scalar, `mrdivide` is equivalent to `rdivide`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

b — Denominator
scalar | vector | matrix | multidimensional array

Denominator, specified as a real scalar, vector, matrix, or multidimensional array. If one or both of the inputs is a `fi` object, then `b` must be a scalar. When `b` is a scalar, `mrdivide` is equivalent to `rdivide`.

If neither input is a `fi` object, then the sizes of the input matrices must be compatible for matrix division.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Output Arguments

X — Quotient
scalar | vector | matrix | multidimensional array

Solution, returned as an array with the same dimensions as the numerator input `A`. When `A` is complex, the real and imaginary parts of `A` are independently divided by `b`.

Version History

Introduced in R2009a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`add` | `divide` | `fi` | `fimath` | `numerictype` | `rdivide` | `sub` | `sum`

mtimes

Matrix product of fi objects

Syntax

```
mtimes(a,b)
```

Description

mtimes(a,b) is called for the syntax a * b when a or b is an object.

a * b is the matrix product of a and b. A scalar value (a 1-by-1 matrix) can multiply any other value. Otherwise, the number of columns of a must equal the number of rows of b.

mtimes does not support fi objects of data type Boolean.

Note For information about the fimath properties involved in Fixed-Point Designer calculations, see “fimath Properties Usage for Fixed-Point Arithmetic” and “fimath ProductMode and SumMode”.

For information about calculations using Fixed-Point Designer software, see the Fixed-Point Designer documentation.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any non-fi input must be constant; that is, its value must be known at compile time so that it can be cast to a fi object.
- Variable-sized inputs are only supported when the SumMode property of the governing fimath is set to SpecifyPrecision or KeepLSB.
- For variable-sized signals, you may see different results between the generated code and MATLAB.

  - In the generated code, the output for variable-sized signals is computed using the SumMode property of the governing fimath.
  - In MATLAB, the output for variable-sized signals is computed using the SumMode property of the governing fimath when both inputs are nonscalar. However, if either input is a scalar, MATLAB computes the output using the ProductMode of the governing fimath.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`plus` | `minus` | `times` | `uminus`

ne, ~=

Package: `embedded`

Determine whether real-world values of two arrays are not equal

Syntax

```
A ~= B
ne(A,B)
```

Description

`A ~= B` returns a logical array with elements set to logical `1` (`true`) where the real-world values of `A` and `B` are not equal, when `A` or `B` is a `fi` object. Otherwise, the element is logical `0` (`false`). The test compares both real and imaginary parts of numeric arrays.

In relational operations comparing a floating-point value to a fixed-point value, the floating-point value is cast to a fixed-point type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object.

`ne(A,B)` is an alternate way to execute `A ~= B`, but is rarely used.

Examples

Compare Two `fi` Objects

Use the `ne` function to determine whether the real-world values of two `fi` objects are not equal.

```
a = fi(pi);
b = fi(pi, 1, 32);
a ~= b
```

```
ans = logical
   1
```

Input `a` has a 16-bit word length, while input `b` has a 32-bit word length. The `ne` function returns `1` because after quantization, the value of `a` is greater than that of `b`.

Compare a Double to a `fi` Object

When comparing a double to a `fi` object, the floating-point double is cast to a type that preserves the relative *order* of the value with respect to the value in the fixed-point `fi` object. This behavior allows relational operations to work between `fi` objects and floating-point constants without introducing floating-point values in generated code.

```
a = fi(pi);
b = pi;
ne(a,b)

ans =

  logical

   1
```

Input Arguments

A,B — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Version History

Introduced before R2006a

R2022a: Implicit expansion change affects arguments for operators
Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for fi ne, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a try/catch block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

R2022a: Improved accuracy in comparing fi objects and floating-point numbers using relational operators
Behavior changed in R2022a

In previous releases, when comparing a single or double to a fi object, the floating-point value was cast to the same word length and signedness of the fi object. This could lead to incorrect results. For example,

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   0   0
```

```
fi(65534)
fi(65534.25) == 65534.25

ans =

       65534

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: -1

ans =

  logical

   1
```

Starting in R2022a, relational operators comparing `fi` objects to floating-point numbers will always return the mathematically correct behavior. The previous examples now gives these results:

```
fi(0,0,8) > [-1,10]

ans =

  1×2 logical array

   1   0
```

Note that the updated algorithm may produce subtle, but accurate, results. For example:

```
fi(pi) == pi

ans =

  logical

   0
```

Simulation results for relational operations between `fi` objects and floating-point singles or doubles may be more accurate than in previous releases. The updated algorithm requires a modest wordlength growth of 3 bits or fewer, which may lead to slight changes in efficiency in simulation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals with different biases are not supported.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

eq | ge | gt | le | lt

nearest

Round toward nearest integer with ties rounding toward positive infinity

Syntax

`y = nearest(a)`

Description

`y = nearest(a)` rounds `fi` object `a` to the nearest integer or, in case of a tie, to the nearest integer in the direction of positive infinity, and returns the result in `fi` object `y`.

Examples

Use `nearest` on a Signed `fi` Object

The following example demonstrates how the `nearest` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 3.

```
a = fi(pi,1,8,3)
```

```
a = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

```
y = nearest(a)
```

```
y = 
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

The following example demonstrates how the `nearest` function affects the `numerictype` properties of a signed `fi` object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a = 
    0.0249

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
y = nearest(a)

y = 
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

Compare Rounding Methods

The functions `convergent`, `nearest`, and `round` differ in the way they treat values whose least significant digit is 5.

- The `convergent` function rounds ties to the nearest even integer.
- The `nearest` function rounds ties to the nearest integer toward positive infinity.
- The `round` function rounds ties to the nearest integer with greater absolute value.

This example illustrates these differences for a given input, a.

```
a = fi([-3.5:3.5]');
y = [a convergent(a) nearest(a) round(a)]

y = 
   -3.5000   -4.0000   -3.0000   -4.0000
   -2.5000   -2.0000   -2.0000   -3.0000
   -1.5000   -2.0000   -1.0000   -2.0000
   -0.5000         0         0   -1.0000
    0.5000         0    1.0000    1.0000
    1.5000    2.0000    2.0000    2.0000
    2.5000    2.0000    3.0000    3.0000
    3.5000    3.9999    3.9999    3.9999

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

a — Input `fi` array
scalar | vector | matrix | multidimensional array

Input `fi` array, specified as scalar, vector, matrix, or multidimensional array.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`nearest` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Data Types: `fi`
Complex Number Support: Yes

Algorithms

- `y` and `a` have the same `fimath` object and `DataType` property.
- When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

Version History

Introduced in R2008a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`ceil` | `convergent` | `fix` | `floor` | `round`

nearestDiv

Round the result of division toward the nearest integer

Syntax

```
y = nearestDiv(x,d)
y = nearestDiv(x,d,m)
```

Description

`y = nearestDiv(x,d)` returns the result of `x/d` rounded to the nearest integer value.

`y = nearestDiv(x,d,m)` returns the result of `x/d` rounded to the nearest multiple of `m`.

The datatype of `y` is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of `x`, and the values of `d` and `m`.

Examples

Divide and Round to Nearest

Perform a division operation and round to the nearest integer value.

```
nearestDiv(int16(201),10)
```

```
ans =
    20

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Perform a division operation and round to the nearest multiple of 7.

```
nearestDiv(int16(201),10,7)
```

```
ans =
    21

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 13
        FractionLength: 0
```

Divide and Generate Code

Define a function that uses nearest`Div`.

```
function y = nearestDiv_example(x,d)
y = nearestDiv(x,d);
end
```

Define inputs and execute the function in MATLAB®.

```
x = fi(pi);
d = fi(2);
y = nearestDiv_example(x,d)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

To generate code for this function, the denominator d must be defined as a constant.

```
codegen nearestDiv_example -args {x, coder.Constant(d)}
```

```
Code generation successful.
```

Alternatively, you can define the denominator, d, as constant in the body of the code.

```
function y = nearestDiv10(x)
y = nearestDiv(x,10);
end
```

```
x = fi(5*pi);
y = nearestDiv10(x)
```

```
y = 
     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

```
codegen nearestDiv10 -args {x}
```

```
Code generation successful.
```

Input Arguments

x — Dividend
scalar

Dividend, specified as a scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | fi

d — Divisor
scalar

Divisor, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`m` — Value to round to nearest multiple of
1 (default) | scalar

Value to round to nearest multiple of, specified as a scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Output Arguments

`y` — Result of division and round to floor
scalar

Result of division and round to floor, returned as a scalar.

The datatype of `y` is calculated such that the wordlength and fraction length are of a sufficient size to contain both the largest and smallest possible solutions given the data type of `x`, and the values of `d` and `m`.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for fixed-point data types.

To generate code, the denominator `d` must be declared as constant.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also

`ceilDiv` | `fixDiv` | `floorDiv`

nextpow2

Package: `embedded`

Exponent of next higher power of 2 of `fi` object

Syntax

```
P = nextpow2(N)
```

Description

`P = nextpow2(N)` returns the first `P` such that `2.^P >= abs(N)`. By convention, `nextpow2(0)` returns zero.

Examples

Next Power of 2 of `fi` Object

Define a `fi` object and calculate the exponent for the next higher power of 2.

```
N = fi(1000,1,18,2);
P = nextpow2(N)
```

```
P = 

    10

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

Next Power of 2 of `fi` Values

Define a vector of `fi` values and calculate the exponents for the next power of 2 higher than those values.

```
N = fi([1 -2 3 -4 5 9 519],1,16,3,2);
P = nextpow2(N)
```

```
P = 

     1     0     1     2     3     3    10

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
```

```
          WordLength: 5
      FractionLength: 0
```

Input Arguments

N — Input values
scalar | vector | *N*-dimensional array

Input values, specified as a real-valued scalar, vector, or *N*-dimensional array.

Data Types: `fi`

Output Arguments

P — Exponent of next higher power of 2
scalar | vector | *N*-dimensional array

Exponent of next higher power of 2, returned as a scalar, vector, or *N*-dimensional array.

The output is returned as an unsigned `fi` object with binary-point scaling, a fraction length of zero, and the smallest word length which can represent the value of the largest returned exponent.

Version History
Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Slope-bias representation is not supported for code generation.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
`nextpow2` | `fi`

nnz

Package: embedded

Number of nonzero elements in fi object

Syntax

```
N = nnz(X)
```

Description

N = nnz(X) returns the number of nonzero elements in X.

When X is a built-in MATLAB type, floating-point fi object, or scaled double fi object, N is returned as a double. When X is a fixed-point fi object, N is returned as a uint32 if X has fewer than 2^{32} elements. Otherwise, N is returned as a uint64.

Examples

Number of Nonzero Elements in fi Object

Create a fi object and determine the number of nonzero elements it contains.

```
p = fi([],1,24,12);
X = eye(2,3,'like',p)
```

```
X =

     1     0     0
     0     1     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

```
N = nnz(X)
```

```
N =

  uint32

   2
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fi` | `nnz`

noperations

Package: `embedded`

Number of quantization operations by `quantizer` object

Syntax

`a = noperations(q)`

Description

`a = noperations(q)` returns the number of quantization operations during a call to `quantize(q,...)` for `quantizer` object `q`. This value accumulates over successive calls to `quantize`. You reset the value of `noperations` to zero by issuing the command `reset(q)` or `resetlog(q)`.

Examples

Count Number of Quantization Operations by Quantizer Object

Create a default `quantizer` object, use it to quantize a vector of values, then return the number of quantization operations performed by the `quantizer` object.

```
q = quantizer;
y = quantize(q,-20:10);
noperations(q)
```

```
Warning: 29 overflow(s) occurred in the fi quantize operation.
> In embedded.quantizer/quantize (line 81)

ans =

    31
```

Input Arguments

q — Input `quantizer` object
`quantizer` object

Input `quantizer` object.

Example: `q = quantizer`

Algorithms

Each time any data element is quantized, `noperations` is incremented by one. The real and complex parts are counted separately. For example, `(complex*complex)` counts four quantization operations for products and two for sum, because`(a+bi)*(c+di) = (a*c - b*d) + (a*d + b*c)`. In contrast, `(real*real)` counts one quantization operation.

In addition, the real and complex parts of the inputs are quantized individually. As a result, for a complex input of length 204 elements, `noperations` counts 408 quantizations: 204 for the real part of the input and 204 for the complex part.

If any inputs, states, or coefficients are complex-valued, they are all expanded from real values to complex values, with a corresponding increase in the number of quantization operations recorded by `noperations`. In concrete terms, `(real*real)` requires fewer quantizations than `(real*complex)` and `(complex*complex)`. Changing all the values to complex because one is complex, such as the coefficient, makes the `(real*real)` into `(real*complex)`, raising `noperations` count.

Version History

Introduced before R2006a

See Also

`quantizer` | `quantize` | `reset` | `resetlog` | `maxlog` | `minlog`

normalizedReciprocal

Compute normalized reciprocal

Syntax

```
[y,e] = normalizedReciprocal(u)
```

Description

`[y,e] = normalizedReciprocal(u)` returns `y` and `e` such that `(2.^e).*y = 1./u` and `0.5 < abs(y) <= 1`.

- If `u = 0` and `u` is a fixed-point or scaled-double data type, then `y = 2 – eps(y)` and `e = 2^(nextpow2(`*`w`*`)) – `*`w`*` + `*`f`*, where *w* is the word length of `u` and *f* is the fraction length of `u`.
- If `u = 0` and `u` is a floating-point data type, then `y = Inf` and `t = 1`.

Examples

Compute Normalized Reciprocal of a Fixed-Point Vector

This example shows how to compute the element-wise normalized reciprocal of a vector of fixed-point values.

```
u = fi([-pi,0.01,pi])
```

```
u =
   -3.1416    0.0100    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
[y,e] = normalizedReciprocal(u)
```

```
y =
   -0.6367    0.7806    0.6367

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14

e = 1x3 int32 row vector

   -1    7   -1
```

Input Arguments

u — Input to take normalized reciprocal of
scalar | vector | matrix | *N*-dimensional array

Input to take the normalized reciprocal of, specified as a real-valued scalar, vector, matrix, or *N*-dimensional array.

Data Types: `single` | `double` | `fi`

Output Arguments

y — Normalized reciprocal
scalar | vector | matrix | *N*-dimensional array

Normalized reciprocal that satisfies `0.5 < abs(y) <= 1` and `(2.^e).*y = 1./u`, returned as a scalar, vector, matrix, or *N*-dimensional array.

- If the input `u` is a signed fixed-point or scaled-double data type with word length *w*, then `y` is a signed fixed-point or scaled-double with word length *w* and fraction length `w - 2`.
- If the input `u` is an unsigned fixed-point or scaled-double data type with word length *w*, then `y` is an unsigned fixed-point or scaled-double with word length *w* and fraction length `w - 1`.
- If the input `u` is a double, then `y` is a double.
- If the input `u` is a single, the `y` is a single.

e — Exponent
scalar | vector | matrix | *N*-dimensional array

Exponent that satisfies `0.5 < abs(y) <= 1` and `(2.^e).*y = 1./u`, returned as an integer scalar, vector, matrix, or *N*-dimensional array.

Version History

Introduced in R2020a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Slope-bias representation is not supported for fixed-point data types.

See Also

Functions
`fi`

Blocks
Normalized Reciprocal HDL Optimized

Topics

“How to Use HDL Optimized Normalized Reciprocal”

noverflows

Number of overflows

Syntax

```
y = noverflows(a)
y = noverflows(q)
```

Description

`y = noverflows(a)` returns the number of overflows of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = noverflows(q)` returns the accumulated number of overflows resulting from quantization operations performed by a `quantizer` object `q`.

Version History

Introduced before R2006a

See Also

`maxlog` | `minlog` | `nunderflows` | `resetlog`

num2bin

Convert number to binary representation using `quantizer` object

Syntax

```
y = num2bin(q,x)
```

Description

`y = num2bin(q,x)` converts the numeric array `x` into a binary character vector returned in `y` using the data type properties specified by the `quantizer` object `q`.

If `x` is a cell array containing numeric matrices, then `x` will be a cell array of the same dimension containing binary strings. If `x` is a structure, then each numeric field of `x` is converted to binary.

[y1,y2,...] = num2bin(q,x1,x2,...) converts the numeric matrices `x1`, `x2`, ... to binary strings `y1`, `y2`, ...

Examples

Convert Numeric Matrix to Binary Character Vector

Convert a matrix of numeric values to a binary character vector using the attributes specified by a `quantizer` object.

```
x = magic(3)/9
```

```
x = 3×3

    0.8889    0.1111    0.6667
    0.3333    0.5556    0.7778
    0.4444    1.0000    0.2222
```

```
q = quantizer([5,3])
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [5  3]
```

```
y = num2bin(q,x)
```

```
y = 9x5 char array
    '00111'
    '00010'
    '00011'
    '00000'
```

```
    '00100'
    '01000'
    '00101'
    '00110'
    '00001'
```

Convert Between Binary String and Numeric Array

Convert between a binary character vector and a numeric array using the properties specified in a `quantizer` object.

Convert Numeric Array to Binary String

Create a `quantizer` object specifying a word length of 4 bits and a fraction length of 3 bits. The other properties of the `quantizer` object take the default values of specifying a signed, fixed-point data type, rounding towards negative infinity, and saturate on overflow.

```
q = quantizer([4 3])
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [4  3]
```

Create an array of numeric values.

```
[a,b] = range(q);
x = (b:-eps(q):a)
```

```
x = 1×16

    0.8750    0.7500    0.6250    0.5000    0.3750    0.2500    0.1250         0   -0.1250   -0.2
```

Convert the numeric vector `x` to binary representation using the properties specified by the `quantizer` object `q`. Note that `num2bin` always returns the binary representations in a column.

```
b = num2bin(q,x)
```

```
b = 16x4 char array
    '0111'
    '0110'
    '0101'
    '0100'
    '0011'
    '0010'
    '0001'
    '0000'
    '1111'
    '1110'
    '1101'
    '1100'
```

```
    '1011'
    '1010'
    '1001'
    '1000'
```

Use `bin2num` to perform the inverse operation.

```
y = bin2num(q,b)
```

```
y = 16×1

    0.8750
    0.7500
    0.6250
    0.5000
    0.3750
    0.2500
    0.1250
         0
   -0.1250
   -0.2500
      ⋮
```

Convert Binary String to Numeric Array

All of the 3-bit fixed-point two's-complement numbers in fractional form are given by:

```
q = quantizer([3 2]);
b = ['011  111'
     '010  110'
     '001  101'
     '000  100'];
```

Use `bin2num` to view the numeric equivalents of these values.

```
x = bin2num(q,b)
```

```
x = 4×2

    0.7500   -0.2500
    0.5000   -0.5000
    0.2500   -0.7500
         0   -1.0000
```

Input Arguments

q — Data type properties to use for conversion
`quantizer` object

Data type properties to use for conversion, specified as a `quantizer` object.

Example: `q = quantizer([16 15]);`

x — Numeric input array
scalar | vector | matrix | multidimensional array | cell array | structure

Numeric input array, specified as a scalar, vector, matrix, multidimensional array, cell array, or structure.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `struct` | `cell`

Tips

- `num2bin` and `bin2num` are inverses of one another. Note that `num2bin` always returns the binary representations in a column.

Algorithms

- The fixed-point binary representation is two's complement.
- The floating-point binary representation is in IEEE Standard 754 style.

Version History

Introduced before R2006a

See Also

`bin2num` | `quantizer` | `hex2num` | `num2hex` | `num2int`

num2hex

Convert number to hexadecimal equivalent using `quantizer` object

Syntax

```
y = num2hex(q,x)
```

Description

`y = num2hex(q,x)` converts numeric matrix `x` into a hexadecimal string returned in `y`. The attributes of the number are specified by the `quantizer` object `q`.

[y1,y2,...] = num2hex(q,x1,x2,...) converts numeric matrices `x1`, `x2`, ... to hexadecimal strings `y1`, `y2`, ...

Examples

Convert Numeric Matrix to Hexadecimal

Use `num2hex` to convert a matrix of numeric values to hexadecimal representation.

Convert Floating-Point Values

This is a floating-point example using a `quantizer` object `q` that has a 6-bit word length and a 3-bit exponent length.

```
x = magic(3);
q = quantizer('float',[6 3]);
y = num2hex(q,x)
```

```
y = 9x2 char array
    '18'
    '12'
    '14'
    '0c'
    '15'
    '18'
    '16'
    '17'
    '10'
```

Convert Fixed-Point Values

All of the 4-bit fixed-point two's complement numbers in fractional form are given by:

```
q = quantizer([4 3]);
x = [0.875    0.375   -0.125   -0.625
     0.750    0.250   -0.250   -0.750
     0.625    0.125   -0.375   -0.875
     0.500        0   -0.500   -1.000];
y = num2hex(q,x)
```

```
y = 16x1 char array
    '7'
    '6'
    '5'
    '4'
    '3'
    '2'
    '1'
    '0'
    'f'
    'e'
    'd'
    'c'
    'b'
    'a'
    '9'
    '8'
```

Input Arguments

q — Attributes of the number
`quantizer` object

Attributes of the number, specified as a `quantizer` object.

x — Numeric values to convert
scalar | vector | matrix | multidimensional array | cell array

Numeric values to convert, specified as a scalar, vector, matrix, multidimensional array, or cell array of doubles.

Data Types: `double` | `cell`
Complex Number Support: Yes

Output Arguments

y — Hexadecimal strings
column vector | cell array

Hexadecimal strings, returned as a column vector. If `x` is a cell array containing numeric matrices, then `y` is returned as a cell array of the same dimension containing hexadecimal strings.

Tips

- `num2hex` and `hex2num` are inverses of each other, except that `hex2num` returns the hexadecimal values in a column.

Algorithms

- For fixed-point `quantizer` objects, the representation is two's complement.
- For floating-point `quantizer` objects, the representation is IEEE Standard 754 style.

  For example, for `q = quantizer('double')`:

```
q = quantizer('double');
num2hex(q,nan)

ans =

    'fff8000000000000'
```

The leading fraction bit is 1, and all the other fraction bits are 0. Sign bit is 1, and exponent bits are all 1.

```
num2hex(q,inf)

ans =

    '7ff0000000000000'
```

Sign bit is 0, exponent bits are all 1, and all fraction bits are 0.

```
num2hex(q,-inf)

ans =

    'fff0000000000000'
```

Sign bit is 1, exponent bits are all 1, and all fraction bits are 0.

Version History

Introduced before R2006a

See Also

bin2num | hex2num | num2bin | num2int | quantizer

num2int

Convert number to signed integer using `quantizer` object

Syntax

```
y = num2int(q,x)
```

Description

`y = num2int(q,x)` converts numeric values in `x` to output `y` containing integers using the data type properties specified by the fixed-point `quantizer` object `q`. If `x` is a cell array containing numeric matrices, then `y` will be a cell array of the same dimension.

[`y1`,`y2`,...] = num2int(`q`,`x1`,`x2`,...) uses `q` to convert numeric values `x1`, `x2`,... to integers `y1`, `y2`,....

Examples

Convert Matrix of Numeric Values to Signed Integer

All the two's complement 4-bit numbers in fractional form are given by:

```
x = [0.875 0.375 -0.125 -0.625
     0.750 0.250 -0.250 -0.750
     0.625 0.125 -0.375 -0.875
     0.500 0.000 -0.500 -1.000];
```

Define a `quantizer` object to use for conversion.

```
q = quantizer([4 3]);
```

Use `num2int` to convert to signed integer.

```
y = num2int(q,x)
```

```
y =

     7     3    -1    -5
     6     2    -2    -6
     5     1    -3    -7
     4     0    -4    -8
```

Input Arguments

q — Data type format to use for conversion
fixed-point `quantizer` object

Data type format to use for conversion, specified as a fixed-point `quantizer` object.

Example: `q = quantizer([5 4]);`

x — Numeric values to convert
scalar | vector | matrix | multidimensional array | cell array

Numeric values to convert, specified as a scalar, vector, matrix, multidimensional array, or cell array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `cell`
Complex Number Support: Yes

Algorithms

- When `q` is a fixed-point `quantizer` object, f is equal to `fractionlength(q)`, and x is numeric:

  $y = x \times 2^f$
- `num2int` is meaningful only for fixed-point `quantizer` objects. When `q` is a floating-point `quantizer` object, x is returned unchanged ($y = x$).
- `y` is returned as a double, but the numeric values will be integers, also known as floating-point integers or flints.

Version History

Introduced before R2006a

See Also

`bin2num` | `hex2num` | `num2bin` | `num2hex` | `quantizer`

num2str

Convert numbers to character array

Syntax

```
s = num2str(A)
s = num2str(A,precision)
s = num2str(A,formatSpec)
```

Description

`s = num2str(A)` converts `fi` object `A` into a character array representation. The output is suitable for input to the `eval` function such that `eval(s)` produces the original `fi` object exactly.

`s = num2str(A,precision)` converts `fi` object `A` to a character array representation using the number of digits of precision specified by `precision`.

`s = num2str(A,formatSpec)` applies a format specified by `formatSpec` to all elements of `A`.

Examples

Convert a fi Object to a Character Vector

Create a `fi` object, `A`, and convert it to a character vector.

```
A = fi(pi)
```

```
A = 

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
S = num2str(A)
```

```
S = 

    '3.1416'
```

Convert a fi Object to a Character with Specified Precision

Create a `fi` object and convert it to a character vector with 8 digits of precision.

```
A = fi(pi)
```

```
A = 
```

```
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
S = num2str(A,8)
```

```
S =

    '3.1416016'
```

Input Arguments

A — Input array
numeric array

Input array, specified as a numeric array.

Data Types: fi | double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
Complex Number Support: Yes

precision — Number of digits of precision
positive integer

Maximum number of significant digits in the output string, specified as a positive integer.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

formatSpec — Format of output fields
formatting operators

Format of the output fields, specified using formatting operators. `formatSpec` also can include ordinary text and special characters.

For more information on formatting operators, see the `num2str` reference page in the MATLAB documentation.

Output Arguments

s — Text representation of input array
character array

Text representation of the input array, returned as a character array.

Version History

Introduced in R2016a

See Also

num2str | mat2str | tostring

numel

Number of data elements in `fi` array

Syntax

```
n = numel(A)
```

Description

`n = numel(A)` returns the number of elements, `n`, in `fi` array `A`.

Using `numel` in your MATLAB code returns the same result for built-in types and `fi` objects. Use `numel` to write data-type independent MATLAB code for array handling.

Examples

Number of Elements in 2-D fi Array

Create a 2-by-3- array of `fi` objects.

```
X = fi(ones(2,3),1,24,12)
```

```
X = 
     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

`numel` counts 6 elements in the matrix.

```
n = numel(X)
```

```
n = 6
```

Number of Elements in Multidimensional fi Array

Create a 2-by-3-by-4 array of `fi` objects.

```
X = fi(ones(2,3,4),1,24,12)
```

```
X = 
(:,:,1) =
     1     1     1
     1     1     1
(:,:,2) =
     1     1     1
```

```
     1     1     1
(:,:,3) =
     1     1     1
     1     1     1
(:,:,4) =
     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

`numel` counts 24 elements in the matrix.

```
n = numel(X)
```

```
n = 24
```

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects.

Complex Number Support: Yes

Version History

Introduced in R2013b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`numel`

numerictype

Construct an `embedded.numerictype` object describing fixed-point or floating-point data type

Syntax

```
T = numerictype
T = numerictype(s)
T = numerictype(s,w)
T = numerictype(s,w,f)
T = numerictype(s,w,slope,bias)
T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)
T = numerictype( ___ ,Name,Value)
T = numerictype(T1,Name,Value)
T = numerictype('Double')
T = numerictype('Single')
T = numerictype('Half')
T = numerictype('Boolean')
```

Description

`T = numerictype` creates a default `numerictype` object.

`T = numerictype(s)` creates a fixed-point `numerictype` object with unspecified scaling, a signed property value of `s`, and a 16-bit word length.

`T = numerictype(s,w)` creates a fixed-point `numerictype` object with unspecified scaling, a signed property value of `s`, and word length of `w`.

`T = numerictype(s,w,f)` creates a fixed-point `numerictype` object with binary point scaling, a signed property value of `s`, word length of `w`, and fraction length of `f`.

`T = numerictype(s,w,slope,bias)` creates a fixed-point `numerictype` object with slope and bias scaling, a signed property value of `s`, word length of `w`, `slope`, and `bias`.

`T = numerictype(s,w,slopeadjustmentfactor,fixedexponent,bias)` creates a fixed-point `numerictype` object with slope and bias scaling, a signed property value of `s`, word length of `w`, `slopeadjustmentfactor`, and `bias`.

`T = numerictype( ___ ,Name,Value)` allows you to set properties using name-value pairs. All properties that you do not specify a value for are assigned their default values.

`T = numerictype(T1,Name,Value)` allows you to make a copy, `T1`, of an existing `numerictype` object, `T`, while modifying any or all of the property values.

`T = numerictype('Double')` creates a `numerictype` object of data type double.

`T = numerictype('Single')` creates a `numerictype` object of data type single.

`T = numerictype('Half')` creates a `numerictype` object of data type half.

`T = numerictype('Boolean')` creates a `numerictype` object of data type Boolean.

Examples

Create a Default numerictype Object

This example shows how to create a `numerictype` object with default property settings.

```
T = numerictype
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Create a numerictype Object with Default Word Length and Scaling

This example shows how to create a `numerictype` object with the default word length and scaling by omitting the arguments for word length, `w`, and fraction length, `f`.

```
T = numerictype(1)
```

```
T =


          DataTypeMode: Fixed-point: unspecified scaling
            Signedness: Signed
            WordLength: 16
```

The object is signed, with a word length of 16 bits and unspecified scaling.

You can use the signedness argument, `s`, to create an unsigned `numerictype` object.

```
T = numerictype(0)
```

```
T =


          DataTypeMode: Fixed-point: unspecified scaling
            Signedness: Unsigned
            WordLength: 16
```

The object is has the default word length of 16 bits and unspecified scaling.

Create a numerictype Object with Unspecified Scaling

This example shows how to create a `numerictype` object with unspecified scaling by omitting the fraction length argument, `f`.

```
T = numerictype(1,32)
```

```
T =


          DataTypeMode: Fixed-point: unspecified scaling
            Signedness: Signed
            WordLength: 32
```

The object is signed, with a 32-bit word length.

Create a `numerictype` Object with Specified Word and Fraction Length

This example shows how to create a signed `numerictype` object with binary-point scaling, a 32-bit word length, and 30-bit fraction length.

```
T = numerictype(1,32,30)
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Create a `numerictype` Object with Slope and Bias Scaling

This example shows how to create a `numerictype` object with slope and bias scaling. The real-world value of a slope and bias scaled number is represented by:

$$\text{realworldvalue} = (\text{slope} \times \text{integer}) + \text{bias}$$

Create a `numerictype` object that describes a signed, fixed-point data type with a word length of 16 bits, a slope of 2^-2, and a bias of 4.

```
T = numerictype(1,16,2^-2,4)
```

```
T =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.25
                  Bias: 4
```

Alternatively, the slope can be represented by:

$$\text{slope} = \text{slopeadjustmentfactor} \times 2^{\text{fixedexponent}}$$

Create a `numerictype` object that describes a signed, fixed-point data type with a word length of 16 bits, a slope adjustment factor of 1, a fixed exponent of -2, and a bias of 4.

```
T = numerictype(1,16,1,-2,4)
```

```
T =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.25
                  Bias: 4
```

Create a numerictype Object with Specified Property Values

This example shows how to use name-value pairs to set `numerictype` properties at object creation.

```
T = numerictype('Signed',true,...
    'DataTypeMode',...
    'Fixed-point: slope and bias scaling', ...
    'WordLength',32,...
    'Slope',2^-2,...
    'Bias',4)
```

```
T =


          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 32
                 Slope: 0.25
                  Bias: 4
```

Create a numerictype Object with Unspecified Sign

This example shows how to create a `numerictype` object with an unspecified sign by using name-value pairs to set the `Signedness` property to `Auto`.

```
T = numerictype('Signedness','Auto')
```

```
T =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Auto
            WordLength: 16
        FractionLength: 15
```

Create a numerictype Object with Specified Data Type

This example shows how to create a `numerictype` object with a specific data type by using arguments and name-value pairs.

```
T = numerictype(0,24,12,'DataType','ScaledDouble')
```

```
T =


          DataTypeMode: Scaled double: binary point scaling
            Signedness: Unsigned
            WordLength: 24
        FractionLength: 12
```

The returned `numerictype` object, `T`, is unsigned, and has a word length of 24 bits, a fraction length of 12 bits, and a data type set to scaled double.

Create a Double, Single, Half, or Boolean numerictype Object

This example shows how to create a `numerictype` object with data type set to double, single, half, or Boolean at object creation.

Create a `numerictype` object with the data type mode set to double.

```
T = numerictype('Double')
```

```
T =


          DataTypeMode: Double
```

Create a `numerictype` object with the data type mode set to single.

```
T = numerictype('Single')
```

```
T =


          DataTypeMode: Single
```

Create a `numerictype` object with the data type mode set to half.

```
T = numerictype('Half')
```

```
T =


          DataTypeMode: Half
```

Create a `numerictype` object with the data type mode set to Boolean.

```
T = numerictype('Boolean')
```

```
T =


          DataTypeMode: Boolean
```

Input Arguments

s — Whether object is signed
true or 1 (default) | false or 0

Whether the object is signed, specified as a numeric or logical 1 (true) or 0 (false).

Example: T = numerictype(true)

Data Types: logical

w — Word length
16 (default) | positive integer

Word length, in bits, of the stored integer value, specified as a positive integer.

Example: T = numerictype(true,16)

Data Types: half | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

f — Fraction length
15 (default) | integer

Fraction length, in bits, of the stored integer value, specified as an integer.

Fraction length can be greater than word length. For more information, see “Binary Point Interpretation” (Fixed-Point Designer).

Example: T = numerictype(true,16,15)

Data Types: half | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

slope — Slope
3.0518e-05 (default) | finite floating-point number greater than zero

Slope, specified as a finite floating-point number greater than zero.

The slope and the bias determine the scaling of a fixed-point number.

Note

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Changing one of these properties affects the others.

Example: T = numerictype(true,16,2^-2,4)

Data Types: half | single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

bias — Bias associated with object
0 (default) | floating-point number

Bias associated with the object, specified as a floating-point number.

The slope and the bias determine the scaling of a fixed-point number.

Example: `T = numerictype(true,16,2^-2,4)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`slopeadjustmentfactor` — Slope adjustment factor
`1` (default) | positive scalar

Slope adjustment factor, specified as a positive scalar.

The slope adjustment factor must be greater than or equal to 1 and less than 2. If you input a `slopeadjustmentfactor` outside this range, the `numerictype` object automatically applies a scaling normalization to the values of `slopeadjustmentfactor` and `fixedexponent` so that the revised slope adjustment factor is greater than or equal to 1 and less than 2, and maintains the value of the slope.

The slope adjustment is equivalent to the fractional slope of a fixed-point number.

Note

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Changing one of these properties affects the others.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`fixedexponent` — Fixed-point exponent
`-15` (default) | integer

Fixed-point exponent associated with the object, specified as an integer.

Note The `FixedExponent` property is the negative of the `FractionLength`. Changing one property changes the other.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `F = numerictype('DataTypeMode','Fixed-point: binary point scaling','DataTypeOverride','Inherit')`

Note When you create a `numerictype` object by using name-value pairs, Fixed-Point Designer creates a default `numerictype` object, and then, for each property name you specify in the constructor, assigns the corresponding value. This behavior differs from the behavior that occurs when you use a syntax such as `T = numerictype(s,w)`. See “Example: Construct a numerictype Object with Property Name and Property Value Pairs”.

Bias — Bias

`0` (default) | floating-point number

Bias, specified as a floating-point number.

The slope and bias determine the scaling of a fixed-point number.

Example: `T = numerictype('DataTypeMode','Fixed-point: slope and bias scaling','Bias',4)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

DataType — Data type category

`'Fixed'` (default) | `'Boolean'` | `'Double'` | `'ScaledDouble'` | `'Single'` | `'Half'`

Data type category, specified as one of these values:

- `'Fixed'` – Fixed-point or integer data type
- `'Boolean'` – Built-in MATLAB Boolean data type
- `'Double'` – Built-in MATLAB double data type
- `'ScaledDouble'` – Scaled double data type
- `'Single'` – Built-in MATLAB single data type
- `'Half'` – MATLAB half-precision data type

Example: `T = numerictype('Double')`

Data Types: `char`

DataTypeMode — Data type and scaling mode

`'Fixed-point: binary point scaling'` (default) | `'Fixed-point: slope and bias scaling'` | `'Fixed-point: unspecified scaling'` | `'Scaled double: binary point scaling'` | `'Scaled double: slope and bias scaling'` | `'Scaled double: unspecified scaling'` | `'Double'` | `'Single'` | `'Half'` | `'Boolean'`

Data type and scaling mode associated with the object, specified as one of these values:

- `'Fixed-point: binary point scaling'` – Fixed-point data type and scaling defined by the word length and fraction length
- `'Fixed-point: slope and bias scaling'` – Fixed-point data type and scaling defined by the slope and bias
- `'Fixed-point: unspecified scaling'` – Fixed-point data type with unspecified scaling
- `'Scaled double: binary point scaling'` – Double data type with fixed-point word length and fraction length information retained
- `'Scaled double: slope and bias scaling'` – Double data type with fixed-point slope and bias information retained
- `'Scaled double: unspecified scaling'` – Double data type with unspecified fixed-point scaling

- `'Double'` – Built-in `double`
- `'Single'` – Built-in `single`
- `'Half'` – MATLAB half-precision data type
- `'Boolean'` – Built-in `boolean`

Example: `T = numerictype('DataTypeMode','Fixed-point: binary point scaling')`

Data Types: `char`

`DataTypeOverride` — Data type override settings
'Inherit' (default) | 'Off'

Data type override settings, specified as one of these values:

- `'Inherit'` – Turn on `DataTypeOverride`
- `'Off'` – Turn off `DataTypeOverride`

Note The `DataTypeOverride` property is not visible when its value is set to the default, `'Inherit'`.

Example: `T = numerictype('DataTypeOverride','Off')`

Data Types: `char`

`FixedExponent` — Fixed-point exponent
`-15` (default) | integer

Fixed-point exponent associated with the object, specified as an integer.

Note The `FixedExponent` property is the negative of the `FractionLength`. Changing one property changes the other.

Example: `T = numerictype('FixedExponent',-12)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FractionLength` — Fraction length of the stored integer value
best precision (default) | integer

Fraction length, in bits, of the stored integer value, specified as an integer.

The default value is the best precision fraction length based on the value of the object and the word length.

Example: `T = numerictype('FractionLength',12)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Scaling` — Fixed-point scaling mode
`'BinaryPoint'` (default) | `'SlopeBias'` | `'Unspecified'`

Fixed-point scaling mode of the object, specified as one of these values:

- 'BinaryPoint' – Scaling for the numerictype object is defined by the fraction length.
- 'SlopeBias' – Scaling for the numerictype object is defined by the slope and bias.
- 'Unspecified' – Temporary setting that is only allowed at numerictype object creation, and allows for the automatic assignment of a best-precision binary point scaling.

Example: `T = numerictype('Scaling','BinaryPoint')`

Data Types: `char`

Signed — Whether the object is signed
`true` or `1` (default) | `false` or `0`

Whether the object is signed, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Note Although the `Signed` property is still supported, the `Signedness` property always appears in the `numerictype` object display. If you choose to change or set the signedness of your `numerictype` object using the `Signed` property, MATLAB updates the corresponding value of the `Signedness` property.

Example: `T = numerictype('Signed',true)`

Data Types: `logical`

Signedness — Whether the object is signed
`'Signed'` (default) | `'Unsigned'` | `'Auto'`

Whether the object is signed, specified as one of these values:

- 'Signed' – Signed
- 'Unsigned' – Unsigned
- 'Auto' – Unspecified sign

Note Although you can create `numerictype` objects with an unspecified sign (`Signedness: Auto`), all fixed-point `numerictype` objects must have a `Signedness` of `Signed` or `Unsigned`. If you use a `numerictype` object with `Signedness: Auto` to construct a `numerictype` object, the `Signedness` property of the `numerictype` object automatically defaults to `Signed`.

Example: `T = numerictype('Signedness','Signed')`

Data Types: `char`

Slope — Slope
`3.0518e-05` (default) | finite, positive floating-point number

Slope, specified as a finite, positive floating-point number.

The slope and bias determine the scaling of a fixed-point number.

Note

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Changing one of these properties affects the others.

Example: `T = numerictype('DataTypeMode','Fixed-point: slope and bias scaling','Slope',2^-2)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`SlopeAdjustmentFactor` — Slope adjustment factor
`1` (default) | positive scalar

Slope adjustment factor, specified as a positive scalar.

The slope adjustment factor must be greater than or equal to 1 and less than 2. If you input a `slopeadjustmentfactor` outside this range, the `numerictype` object automatically applies a scaling normalization to the values of `slopeadjustmentfactor` and `fixedexponent` so that the revised slope adjustment factor is greater than or equal to 1 and less than 2, and maintains the value of the slope.

The slope adjustment is equivalent to the fractional slope of a fixed-point number.

Note

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Changing one of these properties affects the others.

Example: `T = numerictype('DataTypeMode','Fixed-point: slope and bias scaling','SlopeAdjustmentFactor',1.5)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`WordLength` — Word length of the stored integer value
`16` (default) | positive integer

Word length, in bits, of the stored integer value, specified as a positive integer.

Example: `T = numerictype('WordLength',16)`

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced before R2006a

R2021a: Inexact property names for `fi`, `fimath`, and `numerictype` objects not supported

In previous releases, inexact property names for `fi`, `fimath`, and `numerictype` objects would result in a warning. In R2021a, support for inexact property names was removed. Use exact property names instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Fixed-point signals coming in to a MATLAB Function block from Simulink are assigned a `numerictype` object that is populated with the signal's data type and scaling information.
- Returns the data type when the input is a non fixed-point signal.
- Use to create `numerictype` objects in generated code.
- All `numerictype` object properties related to the data type must be constant.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`fi` | `fimath` | `fipref` | `quantizer`

Topics

"numerictype Objects Usage to Share Data Type and Scaling Settings of fi objects"
"numerictype Object Properties"

NumericTypeScope

Determine fixed-point data type

Description

The `NumericTypeScope` object provides information about the dynamic range of your data. The scope provides a visual representation of the dynamic range of your data in the form of a `log2` histogram.

Creation

Syntax

```
H = NumericTypeScope
```

Description

`H = NumericTypeScope` returns a `NumericTypeScope` object. After you create a `NumericTypeScope` object, use the `step` function to process your data and view the `NumericTypeScope`.

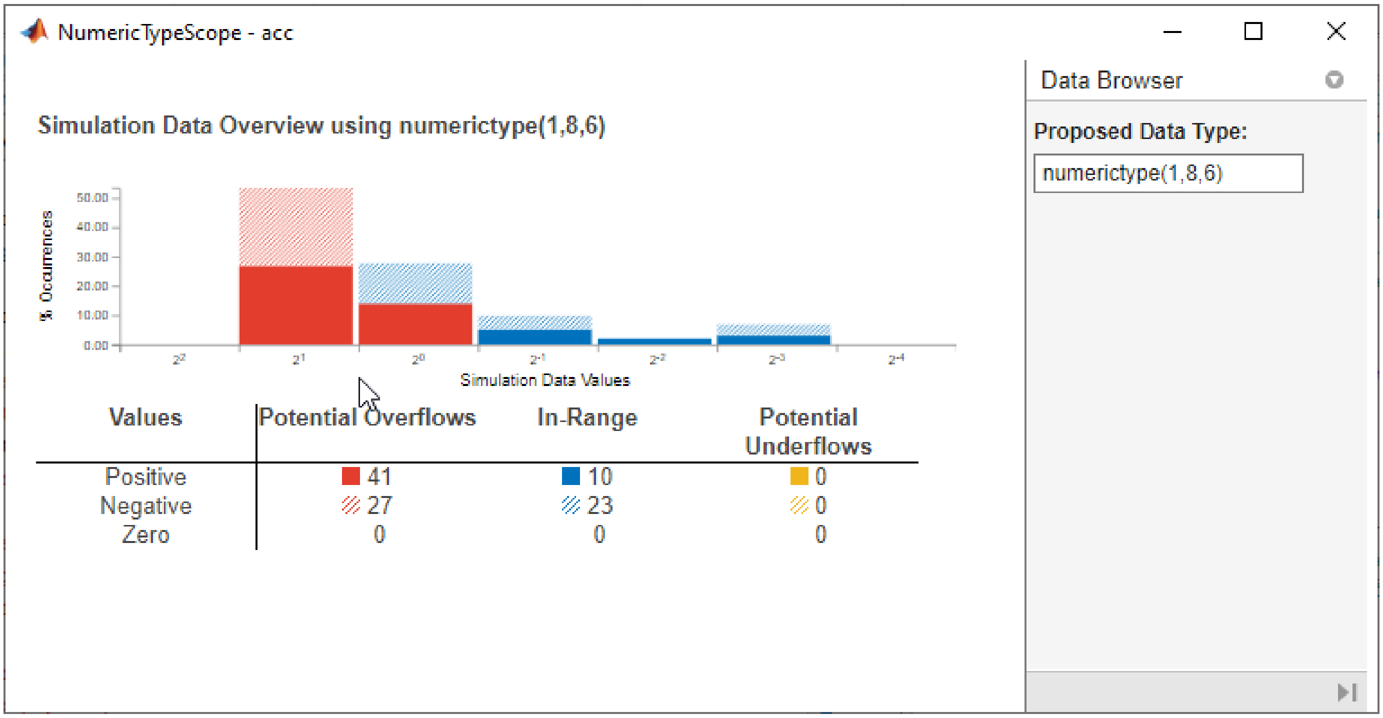
The `NumericTypeScope` window visualizes the dynamic range of a `fi` object in a `log2` histogram. Bit weights appear along the *x*-axis of the histogram and the percentage of occurrences along the *y*-axis. Each bin of the histogram corresponds to a bit in the binary word. For example, 2^0 corresponds to the first integer bit in the binary word, and 2^{-1} corresponds to the first fractional bit in the binary word.

The `NumericTypeScope` identifies potential overflows and underflows based on the current data type. The scope indicates values that may cause overflow or underflow, or are in range of the data type by color-coding the histogram bars as follows:

- Blue — Histogram bin contains values that are in range of the current data type.
- Red — Histogram bin contains values that may cause overflow.
- Yellow — Histogram bin contains values that may cause underflow.

The table below the histogram breaks down each category of values by their signed value.

The **Data Browser** pane displays the current fixed-point data type as the **Proposed Data Type**. You can change the data type by entering a value directly in this box.



Object Functions

step Process data and visualize dynamic range
show Open NumericTypeScope object
reset Clear stored information from NumericTypeScope object

Examples

View the Dynamic Range of a `fi` Object

Use the `NumericTypeScope` to view the dynamic range of a `fi` object.

Create a `fi` object and set the `DataTypeOverride` to `ScaledDoubles`.

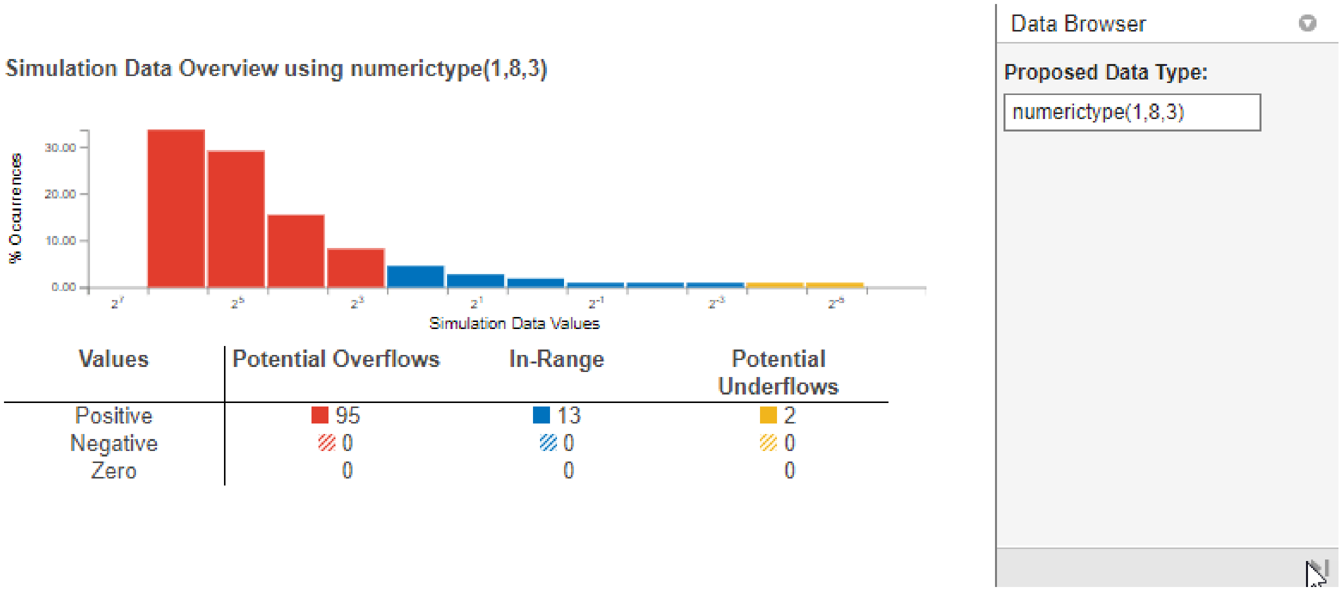
```
a = fi(magic(10),1,8,2);
b = fi([a; 2.^(-5:4)],1,8,3);

fp = fipref;
initialDTOSetting = fp.DataTypeOverride;
fp.DataTypeOverride = 'ScaledDoubles';
```

Create a `NumericTypeScope` object. You can use the `reset` method to ensure that all stored information is cleared from the `NumericTypeScope` object `h`.

```
h = NumericTypeScope;
reset(h)
```

Use the `step` method to process your data and visualize the dynamic range of the `fi` object `b`.

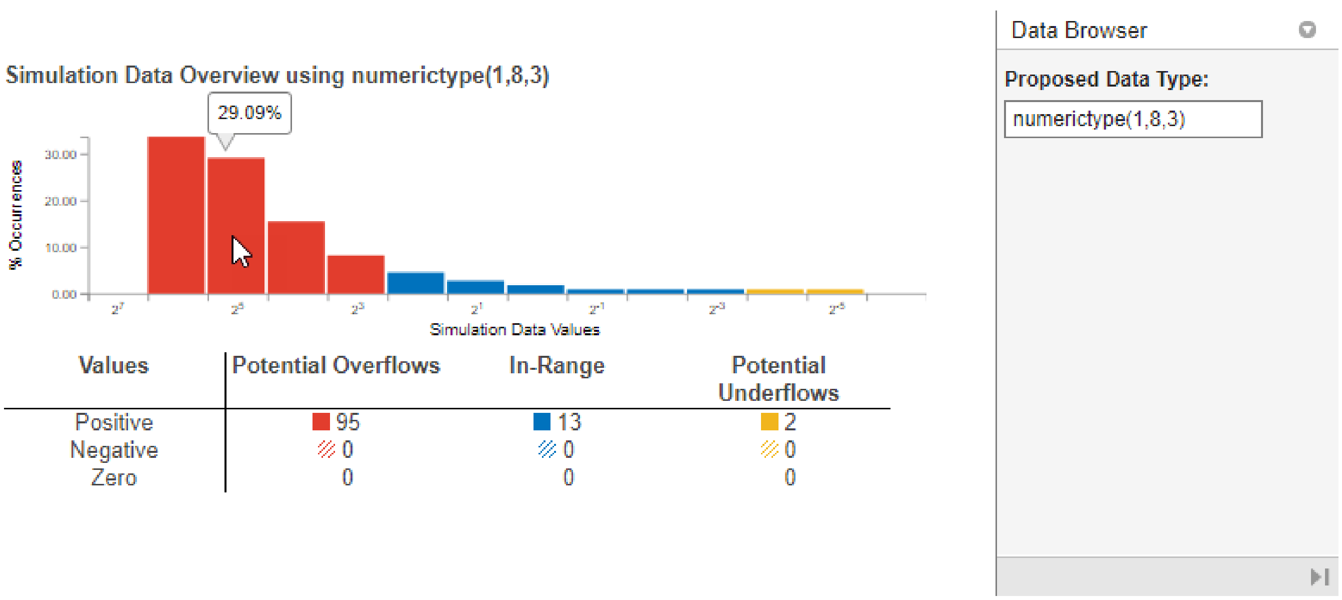
```
step(h,b);
```



Closing the `NumericTypeScope` window does not delete the object from your workspace. Close the `NumericTypeScope` window and reopen it using the `show` function.

```
show(h);
```

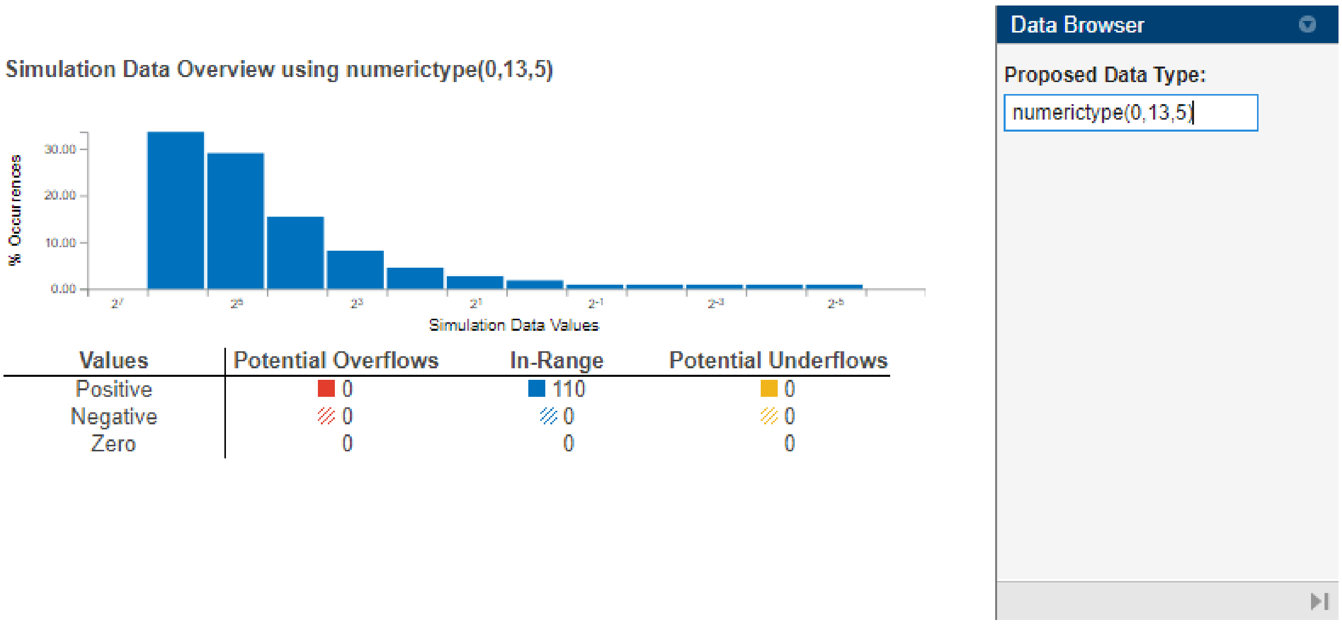
The `NumericTypeScope` displays a `log2` histogram which shows that the values appear both outside of the range and below the precision of the data type of the variable. Pause on one bar of the histogram to view the percentage of the total values that are represented by that bar.



In this case, the data type of `b` is `numerictype(1,8,3)`. The `numerictype(1,8,3)` data type provides 5 integer bits, including the signed bit, and 3 fractional bits. Thus, this data type can represent only values between -2^4 and 2^4 - 2^-3 (from -16 to 15.8750). Given the range and precision of this data type, values greater than 2^4 fall outside the range and values less than 2^-3 fall below the precision of the data type.

The `NumericTypeScope` shows that values requiring bits 5, 6, and 7 are outside the range and values requiring fractional bits 4 and 5 are below precision. Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable `b` to `numerictype(0,13,5)`.

Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable `b` to `numerictype(0,13,5)`. In the `NumericTypeScope`, enter `numerictype(0,13,5)` in the **Proposed Data Type** box.



Return to the original data type override setting.

```
fp.DataTypeOverride = initialDTOSetting;
```

Determine Numeric Type For `fi` Object

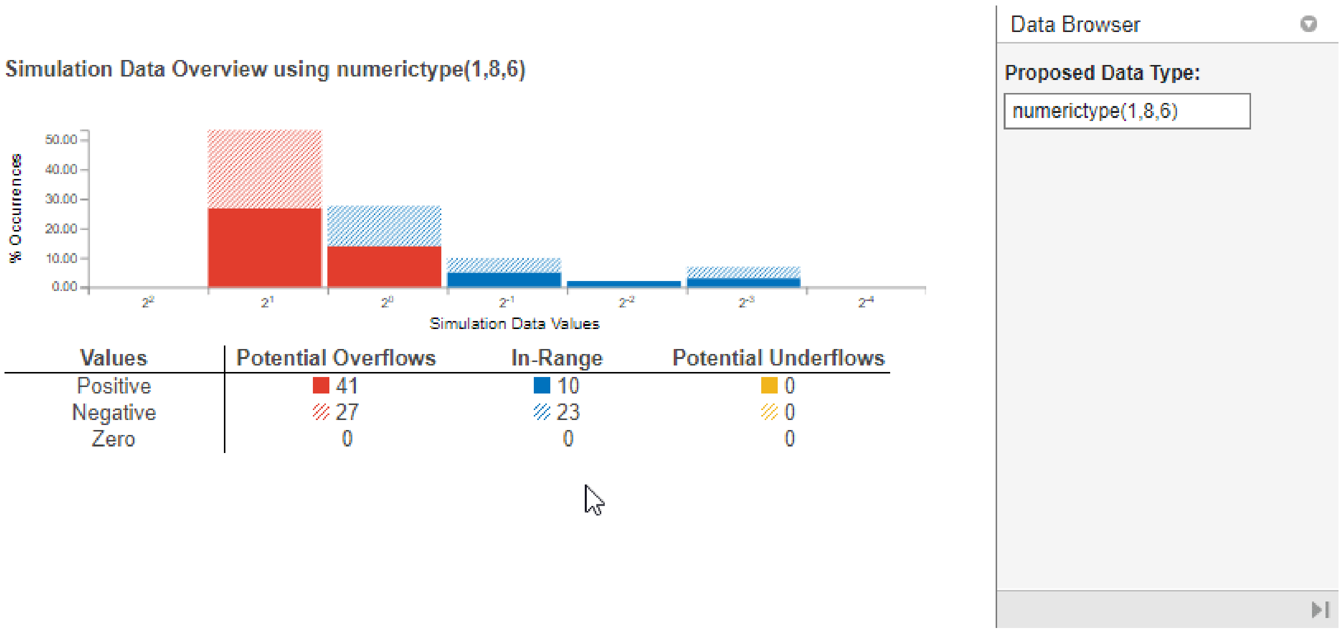
View the dynamic range and determine an appropriate numeric type for a `fi` object with a `DataTypeMode` of `Scaled double: binary point scaling`.

Create a `numerictype` object with a `DataTypeMode` of `Scaled double: binary point scaling`. Then, use that `numerictype` object to construct your `fi` objects.

```
T = numerictype;
T.DataTypeMode = 'Scaled double: binary point scaling';
T.WordLength = 8;
T.FractionLength = 6;

a = fi(sin(0:100)*3.5, T);
b = fi(cos(0:100)*1.75,T);
acc = fi(0,T);
```

Create a `NumericTypeScope` object `h`. Then, use the `step` function in a `for` loop to view the dynamic range of the accumulator object, `acc`.

```
h = NumericTypeScope;
for i = 1:length(a)
    acc(:) = a(i)*0.7+b(i);
    step(h,acc)
end
```

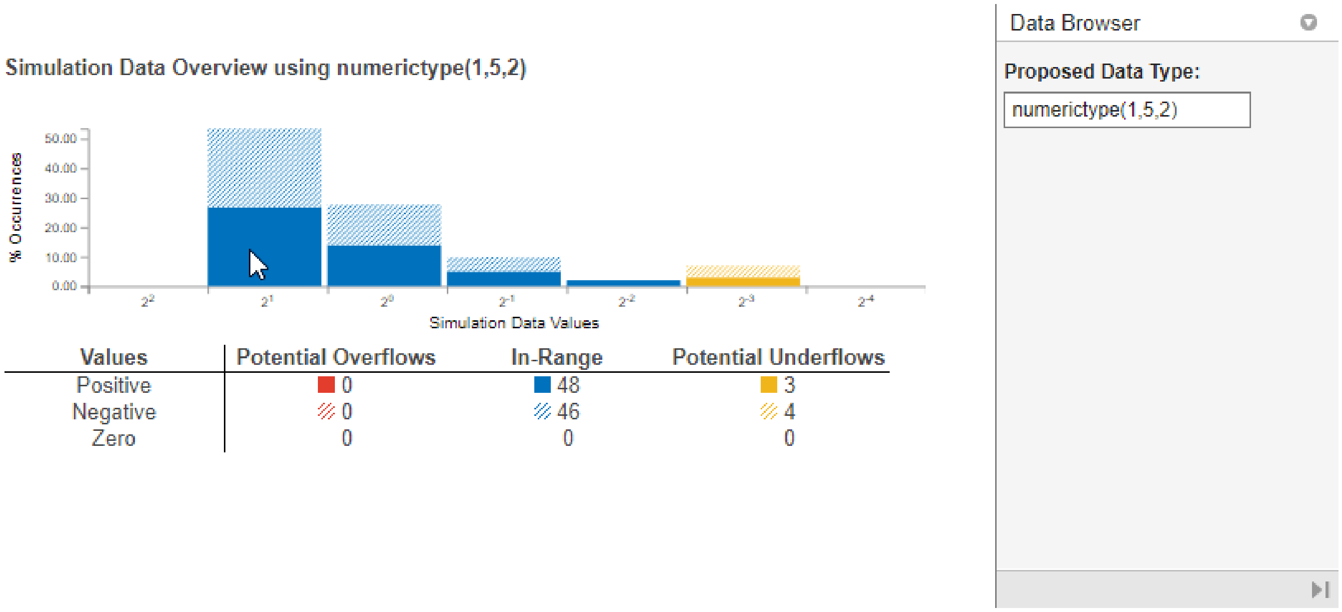


This dynamic range analysis shows that you can represent the entire range of data in the accumulator with 5 bits, two to the left of the binary point (integer bits) and three to the right of it (fractional bits). You can verify that this data type is able to represent all the values by changing the `WordLength` and `FractionLength` properties of the `numerictype` object `T`. Then, use `T` to redefine the accumulator.

```
T.WordLength = 5;
T.FractionLength = 2;
acc = fi(0,T);
```

To view the dynamic range analysis based on this new data type, reset the `NumericTypeScope` object `h`, and rerun the loop.

```
reset(h)
for i = 1:length(a)
    acc(:) = a(i)*0.7 + b(i);
    step(h,acc)
end
```



Clear the information stored in the `NumericTypeScope` object `h`.

```
reset(h);
```

Version History

Introduced in R2010a

R2023a: Updated Programmatic Interface

Behavior changed in R2023a

The `NumericTypeScope`, `step`, `show`, and `reset` functions now launch the updated **Numeric Type Scope** interface. The `release` function has been removed.

R2022b: Updated Numeric Type Scope Interface

Behavior change in future release

The **NumericTypeScope** has a new interface when you launch the scope from the **Instrumentation Report Viewer**. For an example, see `showInstrumentationResults`.

The `NumericTypeScope`, `show`, `step`, `release`, and `reset` functions will be updated in a future release to launch the updated interface.

See Also

`step` | `show` | `reset` | `showInstrumentationResults`

nunderflows

Number of underflows

Syntax

```
y = nunderflows(a)
y = nunderflows(q)
```

Description

`y = nunderflows(a)` returns the number of underflows of `fi` object `a` since logging was turned on or since the last time the log was reset for the object.

Turn on logging by setting the `fipref` property `LoggingMode` to `on`. Reset logging for a `fi` object using the `resetlog` function.

`y = nunderflows(q)` returns the accumulated number of underflows resulting from quantization operations performed by a `quantizer` object `q`.

Version History

Introduced before R2006a

See Also

`maxlog` | `minlog` | `noverflows` | `resetlog`

oct

Package: `embedded`

Octal representation of stored integer of `fi` object

Syntax

`b = oct(a)`

Description

`b = oct(a)` returns the stored integer of `fi` object `a` in octal format as a character vector.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

Tip `oct` returns the octal representation of the stored integer of a `fi` object. To obtain the base-*n* representation of the real-world value of a `fi` object, use `dec2base`.

Examples

View Stored Integer of `fi` Object in Octal Format

Create a signed `fi` object with values `-1` and `1`, a word length of 8 bits, and a fraction length of 7 bits.

```
a = fi([-1 1], 1, 8, 7)
```

```
a = 
   -1.0000    0.9922

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Find the octal representation of the stored integers of `fi` object `a`.

```
b = oct(a)
```

```
b = 
'200   177'
```

Input Arguments

a — Input array
`fi` object

Input array, specified as a `fi` object.

Data Types: `fi`

Version History
Introduced before R2006a

See Also
`bin` | `dec` | `hex` | `storedInteger` | `dec2hex` | `dec2base` | `dec2bin`

ones

Create array of all ones with fixed-point properties

Syntax

```
X = ones('like',p)
X = ones(n,'like',p)
X = ones(sz1,...,szN,'like',p)
X = ones(sz,'like',p)
```

Description

`X = ones('like',p)` returns a scalar `1` with the same `numerictype`, complexity (real or complex), and `fimath` as `p`.

`X = ones(n,'like',p)` returns an `n`-by-`n` array of ones like `p`.

`X = ones(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of ones like `p`.

`X = ones(sz,'like',p)` returns an array of ones like `p`. The size vector, `sz`, defines `size(X)`.

Examples

2-D Array of Ones With Fixed-Point Attributes

Create a 2-by-3 array of ones with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 2-by-3- array of ones that has the same numerictype properties as `p`.

```
X = ones(2,3,'like',p)
```

```
X =
     1     1     1
     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Size Defined by Existing Array

Define a 3-by-2 array `A`.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz = 1×2

     3     2
```

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create an array of ones that is the same size as `A` and has the same numerictype properties as `p`.

```
X = ones(sz,'like',p)
```

```
X = 
     1     1
     1     1
     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Square Array of Ones With Fixed-Point Attributes

Create a 4-by-4 array of ones with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 4-by-4 array of ones that has the same numerictype properties as `p`.

```
X = ones(4, 'like', p)
```

```
X = 
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Create Array of Ones with Attached fimath

Create a signed `fi` object with word length of 16, fraction length of 15 and `OverflowAction` set to `Wrap`.

```
format long
p = fi([],1,16,15,'OverflowAction','Wrap');
```

Create a 2-by-2 array of ones with the same `numerictype` properties as `p`.

```
X = ones(2,'like',p)
```

```
X =
   0.999969482421875   0.999969482421875
   0.999969482421875   0.999969482421875

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15

        RoundingMethod: Nearest
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

1 cannot be represented by the data type of `p`, so the value saturates. The output `fi` object `X` has the same `numerictype` and `fimath` properties as `p`.

Complex Fixed-Point One

Create a scalar fixed-point `1` that is not real valued, but instead is complex like an existing array.

Define a complex `fi` object.

```
p = fi( [1+2i 3i],1,24,12);
```

Create a scalar `1` that is complex like `p`.

```
X = ones('like',p)
```

```
X =
   1.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    for i = 1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818   0.195262145875635   0.097631072937818];
    a = [1.000000000000000  -0.942809041582063   0.333333333333333];

    % Define floating-point types
    T_float.coeffs = double([]);
    T_float.data   = double([]);

    % Create a step input using ones with the
    % floating-point data type
    t = 0:20;
    x_float = ones(size(t),'like',T_float.data);

    % Initialize the states using zeros with the
    % floating-point data type
    z_float = zeros(1,2,'like',T_float.data);

    % Run the floating-point algorithm
    y_float = my_filter(b,a,x_float,z_float,T_float);

    % Define fixed-point types
    T_fixed.coeffs = fi([],true,8,6);
    T_fixed.data   = fi([],true,8,6);

    % Create a step input using ones with the
    % fixed-point data type
    x_fixed = ones(size(t),'like',T_fixed.data);

    % Initialize the states using zeros with the
    % fixed-point data type
    z_fixed = zeros(1,2,'like',T_fixed.data);

    % Run the fixed-point algorithm
    y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

    % Compare the results
```

```
    coder.extrinsic('clf','subplot','plot','legend')
    clf
    subplot(211)
    plot(t,y_float,'co-',t,y_fixed,'kx-')
    legend('Floating-point output','Fixed-point output')
    title('Step response')
    subplot(212)
    plot(t,y_float - double(y_fixed),'rs-')
    legend('Error')
    figure(gcf)
end
```

Input Arguments

n — Size of square matrix
integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If n is zero, X is an empty matrix.
- If n is negative, it is treated as zero.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz1,...,szN — Size of each dimension
two or more integer values

Size of each dimension, specified as two or more integer values, defines X as a sz1-by...-by-szN array.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Example: sz = [2,3,4] defines X as a 2-by-3-by-4 array.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

p — Prototype
fi object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.

If the value 1 overflows the numeric type of `p`, the output saturates regardless of the specified `OverflowAction` property of the attached `fimath`. All subsequent operations performed on the output obey the rules of the attached `fimath`.

Complex Number Support: Yes

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Version History

Introduced in R2013a

See Also

`zeros` | `cast` | `ones`

Topics

"Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types Using cast and zeros"
"Manual Fixed-Point Conversion Workflow"
"Manual Fixed-Point Conversion Best Practices"

plus, +

Package: `embedded`

Matrix sum of `fi` objects

Syntax

```
C = A + B
C = plus(A,B)
```

Description

`C = A + B` adds the matrix `A` to matrix `B`.

`plus` does not support `fi` objects of data type `boolean`.

`C = plus(A,B)` is an alternate way to execute `A + B`.

Note For information about the `fimath` properties involved in Fixed-Point Designer calculations, see "fimath Properties Usage for Fixed-Point Arithmetic" and "fimath ProductMode and SumMode".

Examples

Use Implicit Expansion to Add Vectors, Matrices, and Multidimensional Arrays

This example shows how to use implicit expansion to add vectors and matrices with compatible dimensions.

Add Row and Column Vectors

Create a 3-by-1 column vector and 1-by-5 row vector and add them.

```
x = fi([1;2;3]);
y = fi([1,2,3,4,5]);
z = x + y
```

```
z = 
     2     3     4     5     6
     3     4     5     6     7
     4     5     6     7     8

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 13
```

The result is a 3-by-5 matrix, where each `(i,j)` element in the matrix is given by `z(i,j) = x(i) + y(j)`.

Add Matrix and Column Vector

Create an M-by-N matrix and a M-by-1 column vector and add them.

```
x = fi([1 2 3 4 5
        6 7 8 9 10
        11 12 13 14 15]);
y = fi([1;2;3]);
z = x + y
```

```
z =
     2     3     4     5     6
     8     9    10    11    12
    14    15    16    17    18

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 19
        FractionLength: 13
```

The result is an M-by-N matrix, where each `(i,j)` element in the matrix is given by `z(i,j) = x(i,j) + y(i)`.

Add Matrix and Row Vector

Create a M-by-N matrix and a 1-by-N row vector and add them.

```
x = fi([1 2 3 4 5
        6 7 8 9 10
        11 12 13 14 15]);
y = fi([1 2 3 4 5]);
z = x + y
```

```
z =
     2     4     6     8    10
     7     9    11    13    15
    12    14    16    18    20

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 12
```

The result is an M-by-N matrix, where each (i,j) element in the matrix is given by z(i,j) = x(i,j) + y(j).

Add Matrix to Multidimensional Array

Create a M-by-N matrix and a M-by-N-by-P array and add them.

```
x = fi(ones(3,5));
y = fi(ones(3,5,3));
z = x + y
```

```
z =
(:,:,1) =
     2     2     2     2     2
     2     2     2     2     2
     2     2     2     2     2
(:,:,2) =
```

```
     2     2     2     2     2
     2     2     2     2     2
     2     2     2     2     2
(:,:,3) =
     2     2     2     2     2
     2     2     2     2     2
     2     2     2     2     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 17
        FractionLength: 14
```

The result is an M-by-N-by-P array, where each (i,j,k) element in the array is given by z(i,j,k) = x(i,j) + y(i,j,k).

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in data types. Inputs `A` and `B` must either be the same size or have sizes that are compatible. For more information, see "Compatible Array Sizes for Basic Operations".

`plus` does not support `fi` objects of data type `boolean`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

B — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in data types. Inputs `A` and `B` must either be the same size or have sizes that are compatible. For more information, see "Compatible Array Sizes for Basic Operations".

`plus` does not support `fi` objects of data type `boolean`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

R2021b: Implicit expansion change affects arguments for operators
Behavior changed in R2021b

Starting in R2021b with the addition of implicit expansion for `fi` `times`, `plus`, and `minus`, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any non-`fi` inputs must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Inputs cannot be of data type `logical`.

See Also

`minus` | `mtimes` | `times` | `uminus`

pow10

Base 10 power and scale half-precision numbers

Syntax

`Y = pow10(X)`

Description

`Y = pow10(X)` returns an array, `Y`, whose elements are 10 raised to the power `X`.

Note This function supports only half-precision inputs.

Examples

Base 10 Power

Create a half-precision vector, *X*.

```
X = half([1;2;3;4])
```

```
X =

  4x1 half column vector

     1
     2
     3
     4
```

Compute an array, *Y*, whose elements are 10 raised to the power *X*.

```
Y = pow10(X)
```

```
Y =

  4x1 half column vector

          10
         100
        1000
       10000
```

Input Arguments

X — Power
scalar | vector | matrix | multidimensional array

Power, specified as a half-precision numeric scalar, vector, matrix, or multidimensional array

Data Types: `Half`

Output Arguments

Y — Output array
scalar | vector | matrix | multidimensional array

Array whose elements are 10 raised to the power `X`, returned as a half-precision scalar, vector, matrix, or multidimensional array.

Version History

Introduced in R2018b

See Also

`half`

pow2

Efficient fixed-point multiplication by 2^K

Syntax

`b = pow2(a,K)`

Description

`b = pow2(a,K)` returns the value of `a` shifted by `K` bits where `K` is an integer and `a` and `b` are `fi` objects. The output `b` always has the same word length and fraction length as the input `a`.

Note In fixed-point arithmetic, shifting by `K` bits is equivalent to, and more efficient than, computing $b = a*2^K$.

If `K` is a non-integer, the `pow2` function will round it to `floor` before performing the calculation.

The scaling of `a` must be equivalent to binary point-only scaling; in other words, it must have a power of 2 slope and a bias of 0.

`a` can be real or complex. If `a` is complex, `pow2` operates on both the real and complex portions of `a`.

The `pow2` function obeys the `OverflowAction` and `RoundingMethod` properties associated with `a`. If obeying the `RoundingMethod` property associated with `a` is not important, try using the `bitshift` function.

The `pow2` function does not support `fi` objects of data type `Boolean`.

The function also does not support the syntax `b = pow2(a)` when `a` is a `fi` object.

Examples

Example 4.1. Example 1

In the following example, `a` is a real-valued `fi` object, and `K` is a positive integer.

The `pow2` function shifts the bits of `a` 3 places to the left, effectively multiplying `a` by 2^3.

```
a = fi(pi,1,16,8)
b = pow2(a,3)
binary_a = bin(a)
binary_b = bin(b)
```

```
a =

                  3.140625

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
```

```
        FractionLength: 8

b =

                    25.125

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8

binary_a =

    '0000001100100100'


binary_b =

    '0001100100100000'
```

Example 4.2. Example 2

In the following example, `a` is a real-valued `fi` object, and `K` is a negative integer.

The `pow2` function shifts the bits of `a` 4 places to the right, effectively multiplying `a` by 2^{-4}.

```
a = fi(pi,1,16,8)
b = pow2(a,-4)
binary_a = bin(a)
binary_b = bin(b)
```

```
a =

                  3.140625

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8

b =

                 0.1953125

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 8

binary_a =

    '0000001100100100'


binary_b =

    '0000000000110010'
```

Example 4.3. Example 3

The following example shows the use of `pow2` with a complex `fi` object:

```
format long g
P = fipref('NumericTypeDisplay', 'short');
a = fi(57 - 2i, 1, 16, 8)

a =

                         57 -                    2i
      numerictype(1,16,8)


pow2(a,2)

ans =

               127.99609375 -                    8i
      numerictype(1,16,8)
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

`bitshift` | `bitsll` | `bitsra` | `bitsrl`

power, .^

Package: `embedded`

Fixed-point element-wise power

Syntax

```
C = A.^B
C = power(A, B)
```

Description

`C = A.^B` raises each element of `A` to the corresponding power in `B`.

`C = power(A, B)` is an alternative way to compute `A.^B`.

Examples

Raise Each Element of a Matrix to a Scalar Power

Create a fixed-point matrix and raise it to a scalar power.

```
A = fi([1, 3; 4, 2])
```

```
A =
     1     3
     4     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

```
C = A.^3
```

```
C =
     1    27
    64     8

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 48
        FractionLength: 36
```

Input Arguments

A — Base
scalar | vector | matrix | multidimensional array

Base, specified as a scalar, vector, matrix, or multidimensional array. Inputs A and B must either be the same size or have sizes that are compatible (for example, A is an *M*-by-*N* matrix and B is a scalar or 1-by-*N* row vector).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

B — Exponent
scalar

Exponent, specified as a non-negative, real, integer-valued scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Output Arguments

C — Power
scalar | vector | matrix | multidimensional array

Power, returned as an array with the same dimensions as the input A. When A has a local `fimath` object, the output C also has the same local `fimath` object. The array power operation is always performed using the default `fimath` settings.

Version History

Introduced in R2010a

R2021a: Improved numerical accuracy and generated code efficiency for `fi` inputs to `power, .^`

Fixed-Point Designer now has improved numerical accuracy for fixed-point inputs to the `power` function in simulation and generated code. Additionally, generated code is more efficient.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When the exponent B is a variable, the `ProductMode` property of the governing `fimath` must be `SpecifyPrecision`.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Both inputs must be scalar, and the exponent input, B, must be a constant integer.

See Also
`power` | `mpower`

qr

Orthogonal-triangular decomposition

Description

The Fixed-Point Designer `qr` function differs from the MATLAB `qr` function as follows:

- The input `A` in `qr(A)` must be a real, signed `fi` object.
- The `qr` function ignores and discards any `fimath` attached to the input. The output is always associated with the default `fimath`.
- Pivoting is not supported for fixed-point inputs. You cannot use the following syntaxes:
  - `[~,~,E] = qr(...)`
  - `qr(A,'vector')`
  - `qr(A,B,'vector')`
- Economy size decomposition is not supported for fixed-point inputs. You cannot use the following syntax: `[Q,R] = qr(A,0)`.
- The least-squares-solution form is not supported for fixed-point inputs. You cannot use the following syntax: `qr(A,B)`.

Refer to the MATLAB `qr` reference page for more information.

Version History

Introduced in R2014a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Topics

"Determine Fixed-Point Types for QR Decomposition"

quantize

Package: `embedded`

Quantize fixed-point numbers

Note `quantize` is not recommended. Use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. For more information, see "Compatibility Considerations".

Syntax

```
y = quantize(x)
y = quantize(x,nt)
y = quantize(x,nt,rm)
y = quantize(x,nt,rm,oa)
```

```
yBP = quantize(x,s)
yBP = quantize(x,s,wl)
yBP = quantize(x,s,wl,fl)
yBP = quantize(x,s,wl,fl,rm)
yBP = quantize(x,s,wl,fl,rm,oa)
```

Description

Quantize Using a numerictype Object

`y = quantize(x)` quantizes the input `x` values using the default settings.

The `numerictype`, rounding method, and overflow action apply only during the quantization. The output `y` does not have an attached `fimath`.

`y = quantize(x,nt)` quantizes `x` to the specified `numerictype`, `nt`.

`y = quantize(x,nt,rm)` quantizes `x` to the specified `numerictype`, `nt` using the specified rounding method, `rm`.

`y = quantize(x,nt,rm,oa)` quantizes `x` to the specified `numerictype`, `nt` using the specified rounding method, `rm`, and overflow action, `oa`.

Quantize by Specifying Numeric Type Properties

`yBP = quantize(x,s)` quantizes `x` to a binary-point scaled fixed-point number with signedness `s`.

`yBP = quantize(x,s,wl)` quantizes `x` to a binary-point scaled fixed-point number with signedness `s` and word length `wl`.

`yBP = quantize(x,s,wl,fl)` quantizes `x` to a binary-point scaled fixed-point number with signedness `s`, word length `wl`, and fraction length `fl`.

`yBP = quantize(x,s,wl,fl,rm)` quantizes `x` to a binary-point scaled fixed-point number with signedness `s`, word length `wl`, and fraction length `fl` using rounding method `rm`.

`yBP = quantize(x,s,wl,fl,rm,oa)` quantizes `x` to a binary-point scaled fixed-point number with signedness `s`, word length `wl`, and fraction length `fl` using rounding method `rm` and overflow action `oa`.

Examples

Quantize Binary-Point Scaled to Binary-Point Scaled Data

Define the input `fi` value to quantize.

```
x_BP = fi(pi)
```

```
x_BP = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Use a `numerictype` Object

Create `numerictype` object which specifies a signed fixed-point data type with 8-bit word length and 4-bit fraction length.

```
ntBP = numerictype(1,8,4);
```

Use the defined `numerictype` object `ntBP` to quantize the input `x_BP` to a binary-point scaled fixed-point data type.

```
yBP1 = quantize(x_BP,ntBP)
```

```
yBP1 = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Specify Numeric Type Properties at the Input

```
yBP2 = quantize(x_BP,1,8,4)
```

```
yBP2 = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Quantize Binary-Point Scaled to Slope-Bias Data

Create a `numerictype` object that specifies a slope-bias scaled fixed-point data type.

```
ntSB = numerictype('Scaling','SlopeBias',...
      'SlopeAdjustmentFactor',1.8,...
      'Bias',1,...
      'FixedExponent',-12);
```

Define the input `fi` value to quantize.

```
x_BP = fi(pi)
```

```
x_BP = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Use the defined `numerictype` `ntSB` to quantize the input `x_BP` to a slope-bias scaled fixed-point data type.

```
ySB1 = quantize(x_BP, ntSB)
```

```
ySB1 = 
    3.1415

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.000439453125
                  Bias: 1
```

Quantize Slope-Bias Scaled to Binary-Point Scaled Data

Define the input `fi` values to quantize.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))
```

```
x_SB = 
    0.8147    0.0975    0.1576
    0.8750    0.2785    0.8750
    0.1270    0.5469    0.8750
    0.8750    0.8750    0.4854
    0.6324    0.8750    0.8003

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3.0517578125e-5
                  Bias: -0.125
```

Use a `numerictype` Object

Create a `numerictype` object `ntBP` that specifies a signed, binary-point scaled fixed-point data type with 8-bit word length and 4-bit fraction length.

```
ntBP = numerictype(1,8,4);
```

Use the defined `numerictype` `ntBP` to quantize the input `x_SB` to a binary-point scaled fixed-point data type. Additionally, round to nearest and saturate on overflow.

```
yBP1 = quantize(x_SB,ntBP,'Nearest','Saturate')
```

```
yBP1 =
    0.8125    0.1250    0.1875
    0.8750    0.2500    0.8750
    0.1250    0.5625    0.8750
    0.8750    0.8750    0.5000
    0.6250    0.8750    0.8125

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Specify Numeric Type Properties at the Input

```
yBP2 = quantize(x_SB,1,8,4,'Nearest','Saturate')
```

```
yBP2 =
    0.8125    0.1250    0.1875
    0.8750    0.2500    0.8750
    0.1250    0.5625    0.8750
    0.8750    0.8750    0.5000
    0.6250    0.8750    0.8125

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Quantize Slope-Bias Scaled to Slope-Bias Scaled Data

Define the input `fi` values to quantize.

```
x_SB = fi(rand(5,3),numerictype('Scaling','SlopeBias','Bias',-0.125))
```

```
x_SB =
    0.8147    0.0975    0.1576
    0.8750    0.2785    0.8750
    0.1270    0.5469    0.8750
    0.8750    0.8750    0.4854
    0.6324    0.8750    0.8003

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3.0517578125e-5
                  Bias: -0.125
```

Create a `numerictype` object which specifies a slope-bias scaled fixed-point data type.

```
ntSB = numerictype('Scaling','SlopeBias', ...
      'SlopeAdjustmentFactor',1.8,'Bias',...
      1,'FixedExponent',-12);
```

Use the defined `numerictype` `ntSB` to quantize the input `x_SB` to a slope-bias scaled fixed-point data type. Additionall, round to ceiling.

```
ySB2 = quantize(x_SB,ntSB,'Ceiling')
```

```
ySB2 = 
    0.8150    0.0978    0.1580
    0.8752    0.2789    0.8752
    0.1272    0.5469    0.8752
    0.8752    0.8752    0.4854
    0.6326    0.8752    0.8005

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.000439453125
                  Bias: 1
```

Quantize Built-in Integer to Binary-Point Scaled Data

Define the input values to quantize.

```
xInt = int8(-16:4:16)
```

```
xInt = 1x9 int8 row vector

   -16   -12    -8    -4     0     4     8    12    16
```

Use a `numerictype` Object

Create a `numerictype` object that specifies a signed binary-point scaled fixed-point data type with 8-bit word length and 4-bit fraction length.

```
ntBP = numerictype(1,8,4);
```

Use the defined `numerictype` `ntBP` to quantize the input `xInt` to a binary-point scaled fixed-point data type.

```
yBP1 = quantize(xInt,ntBP,'Zero')
```

```
yBP1 = 
     0     4    -8    -4     0     4    -8    -4     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Show the range of the quantized output.

```
range(yBP1)
```

```
ans = 
   -8.0000    7.9375
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

The first two and last three values are wrapped because they are outside the representable range of the output type.

Specify Numeric Type Properties at the Input

```
yBP2 = quantize(xInt,1,8,4,'Zero')
```

```
yBP2 =
     0     4    -8    -4     0     4    -8    -4     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4
```

Quantize Built-in Integer to Slope-Bias Data

Define the input values to quantize.

```
xInt = int8(-16:4:16)
```

```
xInt = 1x9 int8 row vector

   -16   -12    -8    -4     0     4     8    12    16
```

Create a `numerictype` object that specifies a slope-bias scaled fixed-point data type.

```
ntSB = numerictype('Scaling','SlopeBias', ...
      'SlopeAdjustmentFactor',1.8,'Bias',...
      1,'FixedExponent',-12);
```

Use the defined `numerictype` `ntSB` to quantize the input `xInt` to a slope-bias scaled fixed-point data type.

```
ySB = quantize(xInt,ntSB,'Round','Saturate')
```

```
ySB =
  -13.4000  -11.9814   -7.9877   -3.9939   -0.0002    3.9936    7.9873   11.9811   15.3996

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 0.000439453125
                  Bias: 1
```

Show the range of the quantized output.

```
range(ySB)
```

```
ans =
  -13.4000   15.3996
```

```
DataTypeMode: Fixed-point: slope and bias scaling
  Signedness: Signed
  WordLength: 16
       Slope: 0.000439453125
        Bias: 1
```

The first and last values saturate because they are at the limits of he representable range of the output type.

Input Arguments

x — Input data to quantize
`fi` object | built-in integer

Input data to quantize, specified as:

- Built-in signed or unsigned integers
- Binary point scaled fixed-point `fi`
- Slope-bias scaled fixed-point `fi`

Although `fi` doubles and `fi` singles are allowed as inputs, they pass through the `quantize` function without being quantized.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

nt — numerictype object
`numerictype(true,16,15)` (default) | `numerictype` object

`numerictype` object that describes a fixed-point data type.

rm — Rounding method to use
`'Floor'` (default) | `'Ceiling'` | `'Convergent'` | `'Nearest'` | `'Round'` | `'Zero'`

Rounding method to use for quantization, specified as one of the following:

- `'Ceiling'` — Round up to the next allowable quantized value.
- `'Convergent'` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit after rounding would be set to 0.
- `'Floor'` — Round down to the next allowable quantized value.
- `'Nearest'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.
- `'Round'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up in absolute value.
- `'Zero'` — Round negative numbers up and positive numbers down to the next allowable quantized value.

Data Types: `char`

oa — Action to take on overflow
'Wrap' (default) | 'Saturate'

Action to take on overflow, specified as one of these values:

- 'Saturate' — Overflows saturate.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers, as specified by the numeric type properties, these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- 'Wrap' — Overflows wrap.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers, as specified by the numeric type properties, these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

Data Types: char

s — Signedness
1 (default) | 0

Signedness of the quantized fixed-point number, specified as 1 (signed) or 0 (unsigned).

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

wl — Word length
16 (default) | positive scalar integer

Word length of the stored integer value of the output data, in bits.

fl — Fraction length
wl-1 (default) | scalar integer

Fraction length of the quantized value, specified as a scalar integer.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Version History
Introduced before R2006a

R2013a: quantize is not recommended
Not recommended starting in R2013a

quantize is not recommended. Use cast, zeros, ones, eye, or subsasgn instead. There are no plans to remove quantize.

Starting in R2013a, use cast, zeros, ones, eye, or subsasgn instead. The cast, zeros, ones, eye, and subsasgn functions can quantize other data types in addition to fi objects and encapsulate type information for quantization in an object rather than as separate input arguments.

<table>
<tr><th>Not Recommended</th><th>Recommended</th></tr>
<tr><td><pre>x_BP = fi(pi);
ntBP = numerictype(1,8,4);
yBP = quantize(x_BP,ntBP)

yBP =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4</pre></td><td><pre>x_BP = fi(pi);
ntBP = fi([],1,8,4);
yBP = cast(x_BP,'like',ntBP)

yBP =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 4</pre></td></tr>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fi` | `numerictype` | `cast` | `zeros`

quantizenumeric

Package: embedded

Quantize numeric data

Syntax

```
y = quantizenumeric(x,s,w,f)
y = quantizenumeric(x,s,w,f,r)
y = quantizenumeric(x,s,w,f,r,o)
```

Description

`y = quantizenumeric(x,s,w,f)` quantizes the value specified in `x` using signedness `s`, word length `w`, and fraction length `f`.

Use `quantizenumeric` when you want to simulate full-precision arithmetic with doubles and then add quantization at certain steps in your algorithm without casting to fixed-point types.

`y = quantizenumeric(x,s,w,f,r)` also specifies rounding mode `r`.

`y = quantizenumeric(x,s,w,f,r,o)` also specifies overflow mode `o`.

Examples

Quantize Value of pi

Quantize the value of pi using a signed numeric type with a word length of 16 bits, a fraction length of 13 bits, and rounding towards positive infinity.

```
x = pi;
y = quantizenumeric(x,1,16,13,'ceil')
```

```
y = 3.1416
```

Specify a different rounding method. Observe that rounding towards zero affects the quantized value.

```
x = pi;
y = quantizenumeric(x,1,16,13,'fix')
```

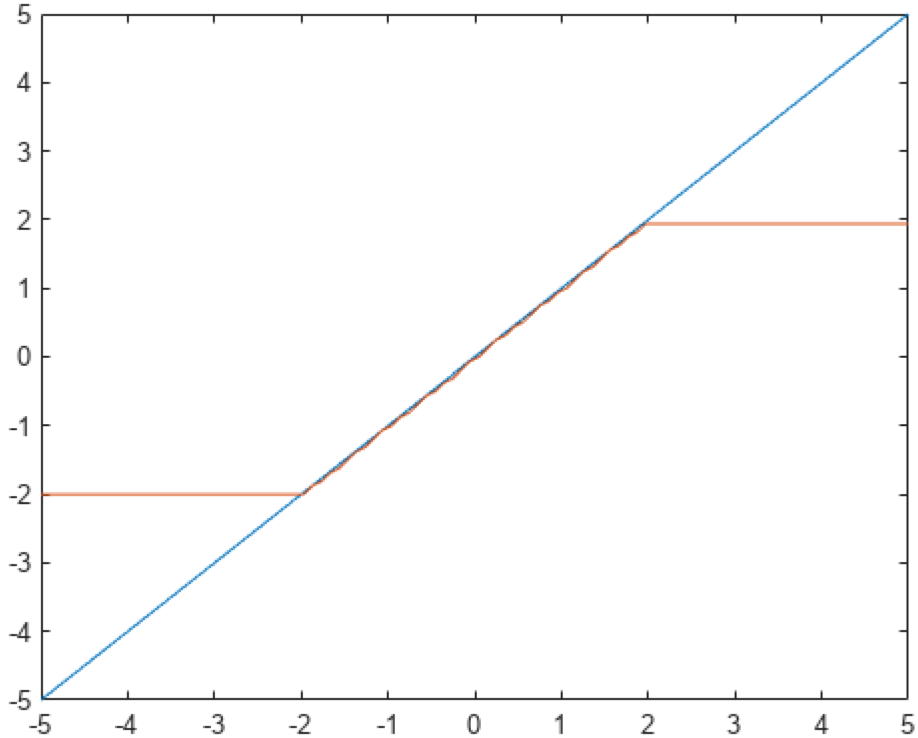
```
y = 3.1415
```

Quantize Numeric Data

This example shows the effect of overflow action on the quantization of numeric data.

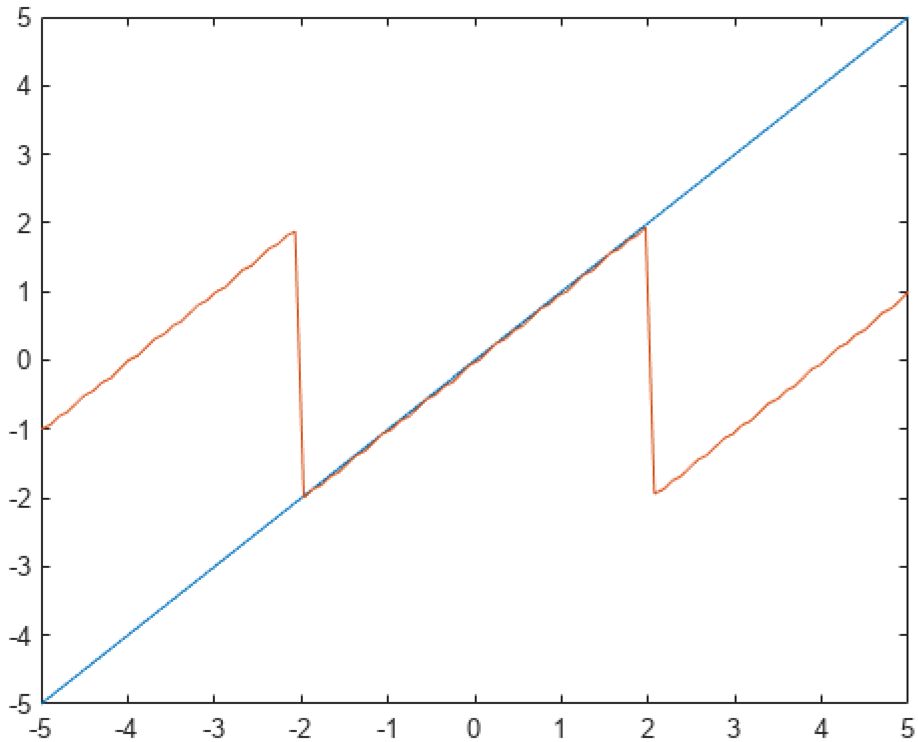
Create some data and quantize it with saturation on overflow specified.

```
x = linspace(-5,5,100);
y = quantizenumeric(x,1,6,4,'floor','saturate');
plot(x,x,x,y)
```



Change the overflow action to wrap on overflow and observe how the quantized data changes.

```
z = quantizenumeric(x,1,6,4,'floor','wrap');
plot(x,x,x,z);
```



Input Arguments

x — Value to quantize
scalar | vector | matrix | multidimensional array

Value to quantize, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `double`
Complex Number Support: Yes

s — Signedness
`0` or `'false'` | `1` or `'true'`

Signedness of quantized value, specified as either `0` or `'false'` (unsigned) or `1` or `'true'` (signed).

Data Types: `double`

w — Word length
positive scalar integer

Word length of quantized value, specified as a positive scalar integer.

Data Types: `double`

f — Fraction length
scalar integer

Fraction length of quantized value, specified as a scalar integer.

Data Types: `double`

r — Rounding method
`'nearest'` (default) | `'ceil'` | `'ceiling'` | `'convergent'` | `'fix'` | `'floor'` | `'round'` | `'zero'`

Rounding method to use for quantization, specified as a character vector:

- `'ceil'` — Round towards positive infinity (same as `'ceiling'`)
- `'ceiling'` — Round towards positive infinity (same as `'ceil'`)
- `'convergent'`— Convergent rounding
- `'fix'`— Round towards zero (same as `'zero'`)
- `'floor'`— Round towards negative infinity
- `'nearest'`— Round towards nearest with ties rounding towards positive infinity
- `'round'`— Round towards nearest with ties rounding up in absolute value
- `'zero'`— Round towards zero (same as `'fix'`)

Data Types: `char`

o — Overflow action
`'saturate'` (default) | `'wrap'`

Overflow action to use for quantization, specified as either `'saturate'` or `'wrap'`.

Data Types: `char`

Output Arguments

y — Quantized output value
scalar | vector | matrix | multidimensional array

Quantized output value, returned as a scalar, vector, matrix, or multidimensional array. `y` always has the same dimensions as `x` and is always a double.

Tips

- Use `quantizenumeric` when you want to simulate full-precision arithmetic with doubles and then add quantization at certain steps in your algorithm without casting to fixed-point types.
- When designing fixed-point algorithms, use `cast`, `zeros`, `ones`, `eye`, and `subsasgn` to separate the core algorithm from data type definitions.

Version History

Introduced in R2016a

R2021b: Change in default behavior of `quantizenumeric` for complex input
Behavior changed in R2021b

In previous releases, `quantizenumeric` would remove the imaginary part of a complex input `x`. For example,

```
x = complex(pi, exp(1))
y = quantizenumeric(x,1,16,12,'floor')
```

```
x =

   3.1416 + 2.7183i


y =

    3.1414
```

`quantizenumeric` now preserves the imaginary part, in the same way as other quantize functions behave for complex inputs. For example,

```
x = complex(pi, exp(1))
y = quantizenumeric(x,1,16,12,'floor')
```

```
x =

   3.1416 + 2.7183i


y =

   3.1414 + 2.7183i
```

See Also

`quantize` | `quantizer` | `cast`

quantize

Package: embedded

Quantize numeric data using quantizer object

Syntax

```
y = quantize(q,x)
[y1,y2,…] = quantize(q,x1,x2,…)
```

Description

y = quantize(q,x) uses the quantizer object q to quantize x.

- When x is a numeric array, each element of x is quantized. The output y is returned as a built-in double.
- When x is a cell array, each numeric element of the cell array is quantized. The fields of output y are returned as built-in doubles.
- When x is a structure, each numeric field of x is quantized. The fields of output y are returned as built-in doubles.

quantize does not change nonnumeric elements or fields of x, nor does it issue warnings for nonnumeric values.

The quantizer object states max, min, noverflows, nunderflows, and noperations are updated during the call to quantize, and running totals are kept until a call to reset is made.

[y1,y2,…] = quantize(q,x1,x2,…) is equivalent to y1 = quantize(q,x1), y2 = quantize(q,x2), … and so forth.

Examples

Quantize Data to Custom-Precision Floating-Point Type

Use quantize to quantize data to a custom-precision floating-point type.

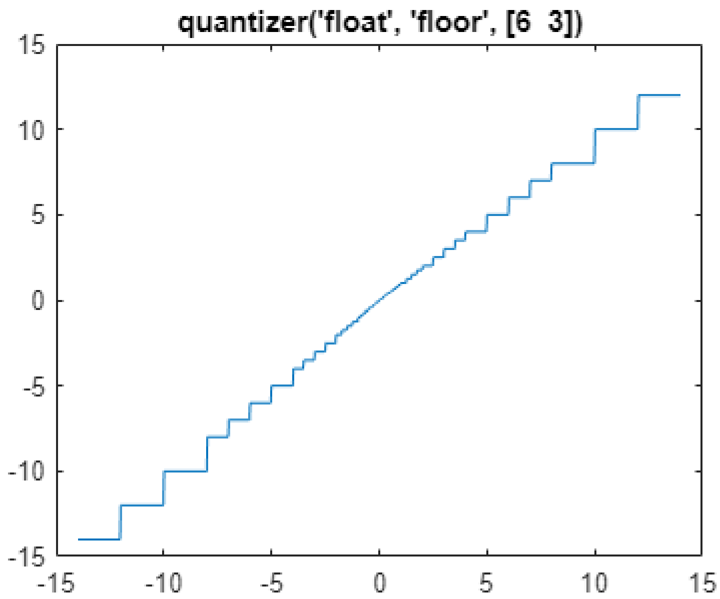
```
x = linspace(-15,15,1000);
q = quantizer('float','floor',[6 3]);
range(q)
```

```
ans = 1×2

   -14    14

```

```
y = quantize(q,x);
```

```
Warning: 68 overflow(s) occurred in the fi quantize operation.
```

```
plot(x,y); title(tostring(q))
```



Quantize to Fixed-Point Type

Use `quantize` to quantize data to a fixed-point type with a wordlength of 6 bits, a fraction length of 2 bits, round to floor, and wrap on overflow.

```
x = linspace(-15,15,1000);
q = quantizer('fixed','floor','wrap',[6 2])
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = wrap
          Format = [6  2]
```

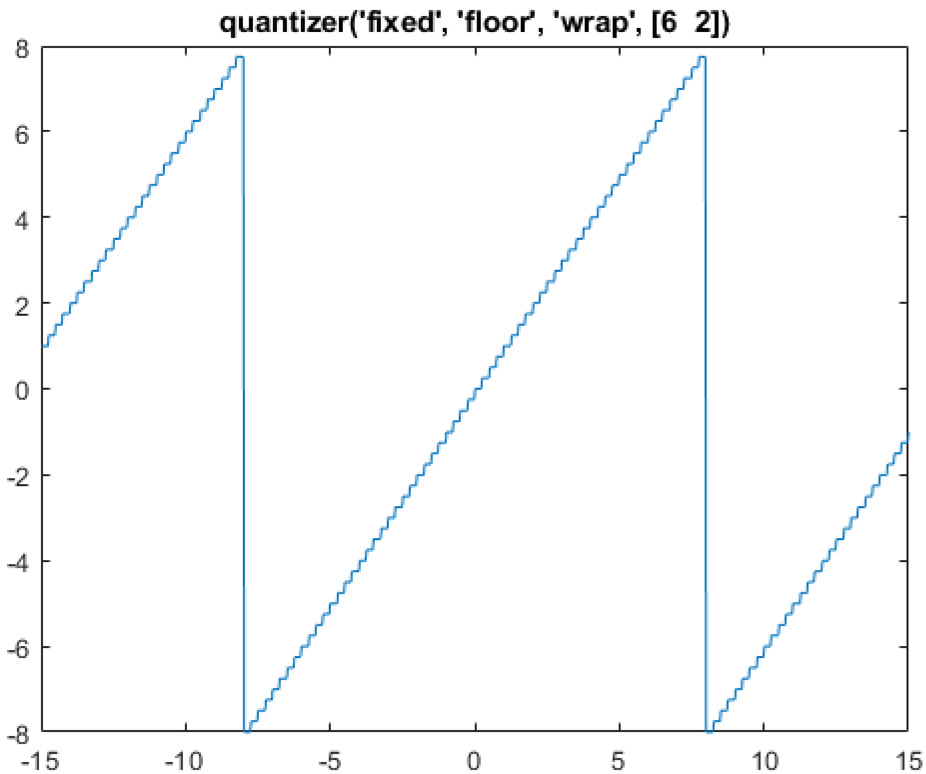
```
range(q)
```

```
ans = 1×2

   -8.0000    7.7500
```

```
y = quantize(q,x);
```

```
Warning: 468 overflow(s) occurred in the fi quantize operation.
```

```
plot(x,y); title(tostring(q))
```

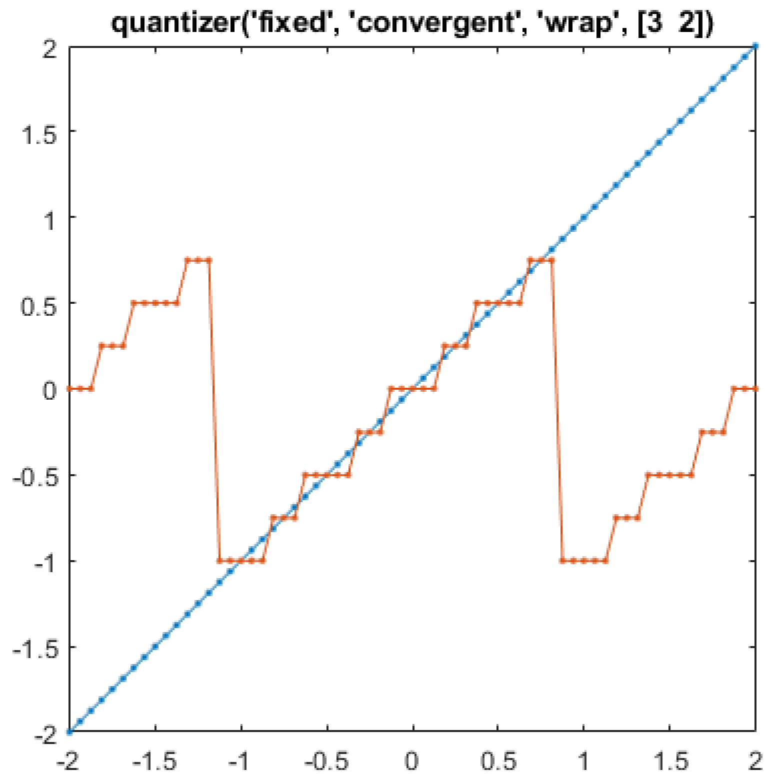


Use `quantize` to quantize data to a fixed-point type with a wordlength of 3 bits, a fraction length of 2 bits, convergent rounding, and wrap on overflow.

```
q = quantizer('fixed','convergent','wrap',[3 2]);
x = (-2:eps(q)/4:2)';
y = quantize(q,x);
```

```
Warning: 33 overflow(s) occurred in the fi quantize operation.
```

```
plot(x,[x,y],'.-'); title(tostring(q)); axis square
```



Input Arguments

q — Data type properties to use for quantization
quantizer object

Data type properties to use for quantization, specified as a `quantizer` object.

Example: `q = quantizer('fixed','ceil','saturate',[5 4]);`

x — Data to quantize
scalar | vector | matrix | multidimensional array | cell array | structure

Data to quantize, specified as a scalar, vector, matrix, multidimensional array, cell array, or structure.

- When `x` is a numeric array, each element of `x` is quantized.
- When `x` is a cell array, each numeric element of the cell array is quantized.
- When `x` is a structure, each numeric field of `x` is quantized.

`quantize` does not change nonnumeric elements or fields of `x`, nor does it issue warnings for nonnumeric values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `struct` | `cell`
Complex Number Support: Yes

x1,x2,… — Data to quantize (as separate elements)
scalar | vector | matrix | multidimensional array | cell array | structure

Data to quantize (as separate elements), specified as a scalar, vector, matrix, multidimensional array, cell array, or structure.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | struct | cell
Complex Number Support: Yes

Output Arguments

y — Quantized data
scalar | vector | matrix | multidimensional array | cell array | structure

Quantized data, returned as a scalar, vector, matrix, multidimensional array, cell array, or structure.

- When x is a numeric array, the output y is returned as a built-in double.
- When x is a cell array, the fields of output y are returned as built-in doubles.
- When x is a structure, the fields of output y are returned as built-in doubles.

[y1,y2,…] — Quantized data (as separate elements)
scalar | vector | matrix | multidimensional array | cell array | structure

Quantized data (as separate elements), returned as a scalar, vector, matrix, multidimensional array, cell array, or structure.

Version History

Introduced in R2012b

R2021b: Change in rounding behavior for quantize function
Behavior changed in R2021b

In previous releases, quantize would round to infinity for values in the range realmax < input < realmax + 0.5*eps(realmax) and negative infinity for values in the range -realmax > x > -realmax - 0.5*eps. Starting in R2021b, values in these ranges quantize as follows, depending on the rounding method used.

Rounding Method	Values in the range realmax < input < realmax + 0.5*eps(realmax) round to	Values in the range -realmax > x > -realmax - 0.5*eps round to
floor	realmax (for x < realmax + eps)	-Inf
ceil	Inf	-realmax (for x > -realmax - eps)
round	realmax	-realmax
convergent	realmax	-realmax
fix	realmax (for x < realmax + eps)	-realmax (for x > -realmax - eps)

Rounding Method	Values in the range realmax < input < realmax + 0.5*eps(realmax) round to	Values in the range -realmax > x > -realmax - 0.5*eps round to
nearest	realmax	-realmax

See Also

quantizer | reset | unitquantize

quantizer

Create quantizer object

Description

The quantizer object describes data type properties to use for quantization. After you create a quantizer object, use quantize to quantize double-precision data. You can use the quantizer object to simulate custom floating-point data types with arbitrary word length and exponent length.

Creation

Syntax

```
q = quantizer
q = quantizer(Name,Value)
q = quantizer(Value1,Value2)
q = quantizer(s)
q = quantizer(pn,pv)
```

Description

q = quantizer creates a quantizer object with properties set to their default values. To use this object to quantize values, use quantize.

q = quantizer(Name,Value) sets named properties using name-value arguments. You can specify multiple name-value arguments. Enclose each property name in single quotes.

q = quantizer(Value1,Value2) sets properties using property values. Property values are unique; you can set the property names by specifying just the property values in the command. When two values conflict, quantizer sets the last property value in the list.

q = quantizer(s) sets properties named in each field name with the values contained in the structure s.

q = quantizer(pn,pv) sets the named properties specified in the cell array of character vectors pn to the corresponding values in the cell array pv.

You can use a combination of name-value string arguments, structures, and name-value cell array arguments to set property values when creating a quantizer object.

Properties

DataMode — Data type mode
'fixed' (default) | 'ufixed' | 'float' | 'single' | 'double'

Data type mode used in quantization, specified as one of these values:

- `'fixed'` — Signed fixed-point mode.
- `'ufixed'` — Unsigned fixed-point mode.
- `'float'` — Custom-precision floating-point mode.
- `'single'` — Single-precision mode. This mode overrides all other property settings.
- `'double'` — Double-precision mode. This mode overrides all other property settings.

Data Types: `char` | `struct` | `cell`

`RoundMode` — Rounding method to use
`'floor'` (default) | `'ceil'` | `'convergent'` | `'fix'` | `'nearest'` | `'round'`

Rounding method to use, specified as one of these values:

- `'ceil'` — Round up to the next allowable quantized value.
- `'convergent'` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit after rounding would be set to 0.
- `'fix'` — Round negative numbers up and positive numbers down to the next allowable quantized value.
- `'floor'` — Round down to the next allowable quantized value.
- `'nearest'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.
- `'round'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up in absolute value.

Data Types: `char` | `struct` | `cell`

`OverflowMode` — Action to take on overflow
`'saturate'` (default) | `'wrap'`

Action to take on overflow, specified as one of these values:

- `'saturate'` — Overflows saturate.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers as specified by the data format properties, these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- `'wrap'` — Overflows wrap to the range of representable values.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers as specified by the data format properties, these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

This property only applies to fixed-point data type modes. This property becomes a read-only property when you set the `DataMode` property to `float`, `double`, or `single`.

Note Floating-point numbers that extend beyond the dynamic range overflow to `±Inf`.

Data Types: `char` | `struct` | `cell`

Format — Data format of quantizer object
[16 15] (default) | [wordlength fractionlength] | [wordlength exponentlength] | [64 11] | [32 8]

Data format of quantizer object. The interpretation of this property value depends on the value of the DataMode property.

DataMode Property Value	Interpreting the Format Property Values
fixed or ufixed	[wordlength fractionlength] Specify the Format property value as a two-element row vector, where the first element is the number of bits for the quantizer object word length and the second element is the number of bits for the quantizer object fraction length. The word length can range from 2 to the limits of memory on your PC. The fraction length can range from 0 to one less than the word length.
float	[wordlength exponentlength] Specify the Format property value as a two-element row vector, where the first element is the number of bits for the quantizer object word length and the second element is the number of bits for the quantizer object exponent length. The word length can range from 2 to the limits of memory on your PC. The fraction length can range from 0 to 11.
double	[64 11] The read-only Format property value automatically specifies the word length and exponent length.
single	[32 8] The read-only Format property value automatically specifies the word length and exponent length.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Read-Only quantizer Object States

Read-only quantizer object states are updated when quantize is called. To reset these states, use reset.

max — Maximum value before quantization
scalar

Maximum value before quantization during a call to quantize(q,…) for quantizer object q. This value is the maximum value recorded over successive calls to quantize.

Example: `max(q)`

Example: `q.max`

`min` — Minimum value before quantization
scalar

Minimum value before quantization during a call to `quantize(q,…)` for `quantizer` object `q`. This value is the minimum value recorded over successive calls to `quantize`.

Example: `min(q)`

Example: `q.min`

`noverflows` — Number of overflows
scalar

Number of overflows during a call to `quantize(q,…)` for `quantizer` object `q`. This value accumulates over successive calls to `quantize`. An overflow is defined as a value that when quantized is outside the range of `q`.

Example: `noverflows(q)`

Example: `q.noverflows`

`nunderflows` — Number of underflows
scalar

Number of underflows during a call to `quantize(q,…)` for `quantizer` object `q`. This value accumulates over successive calls to `quantize`. An underflow is defined as a number that is nonzero before it is quantized and zero after it is quantized.

Example: `nunderflows(q)`

Example: `q.nunderflows`

`noperations` — Number of data points quantized
scalar

Number of quantization operations during a call to `quantize(q,…)` for `quantizer` object `q`. This value accumulates over successive calls to `quantize`.

Example: `noperations(q)`

Example: `q.noperations`

Object Functions

quantize	Quantize numeric data using quantizer object
unitquantize	Quantize numeric data using quantizer object except numbers within eps of +1
wordlength	Word length of quantizer object

Examples

Create `quantizer` Object

Create a `quantizer` object with default property values.

```
q = quantizer
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  15]
```

To copy a `quantizer` object, use assignment.

```
q = quantizer;
r = q;
isequal(q,r)
```

```
ans = logical
   1
```

Use property name-value arguments to set `quantizer` object properties.

```
q = quantizer('Mode','fixed','RoundMode','ceil',...
'OverflowMode','saturate','Format',[5 4])
```

```
q =


        DataMode = fixed
       RoundMode = ceil
    OverflowMode = saturate
          Format = [5  4]
```

Set `quantizer` object properties by listing property values only in the command.

```
q = quantizer('fixed','ceil','saturate',[5 4])
```

```
q =


        DataMode = fixed
       RoundMode = ceil
    OverflowMode = saturate
          Format = [5  4]
```

Use a structure to set `quantizer` object properties.

```
struct.DataMode = 'fixed';
struct.RoundMode = 'ceil';
struct.OverflowMode = 'saturate';
struct.Format = [5 4];
q = quantizer(struct)
```

```
q =


        DataMode = fixed
       RoundMode = ceil
    OverflowMode = saturate
          Format = [5  4]
```

Use property name and property value cell arrays to set `quantizer` object properties.

```
pn = {'Mode','RoundMode','Overflowmode','Format'};
pv = {'fixed','ceil','saturate',[5 4]};
q = quantizer(pn,pv)
```

```
q =


        DataMode = fixed
       RoundMode = ceil
    OverflowMode = saturate
          Format = [5  4]
```

Quantize Data with quantizer Objects

Use `quantize` to quantize data, see how quantization affects `quantizer` object states, and reset `quantizer` object states to their default values using `reset`.

Construct an example data set and create a `quantizer` object to specify the quantization parameters to use when you quantize the data set.

```
format long g
rng(0,'twister');
x = rng(100);
q = quantizer([16,14])
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [16  14]
```

Retrieve the values of `max` and `noverflows`.

```
q.max
q.noverflows
```

```
ans =

    -1.79769313486232e+308


ans =

     0
```

Note that `max` is equal to `-realmax`, which indicates that the quantizer `q` is in a reset state.

Use the `quantize` function to quantize the data set according to the specifications of the `quantizer` object.

```
y = quantize(q,x);
```

```
Warning: 625 overflow(s) occurred in the fi quantize operation.
```

Check the values of max and noverflows.

```
q.max
q.noverflows
```

```
ans =

         1.99993896484375


ans =

   625
```

Note that the maximum logged value was taken after quantization, that is, q.max == max(y).

Reset and check the quantizer states.

```
reset(q)
q.maxlog
q.noverflows
```

```
ans =

    -1.79769313486232e+308


ans =

     0
```

Quantize Data Using the quantizer Object

This example shows how to quantize data using the properties specified by the quantizer object.

First, create some data to quantize.

```
x = linspace(-15,15,1000);
```

Quantize to Custom-Precision Floating-Point

Create a quantizer object specifying a custom-precision floating-point data mode with a word length of 6 bits and an exponent length of 4 bits.

```
q = quantizer('DataMode','float','Format',[6 4])
```

```
q =


        DataMode = float
       RoundMode = floor
          Format = [6  4]
```

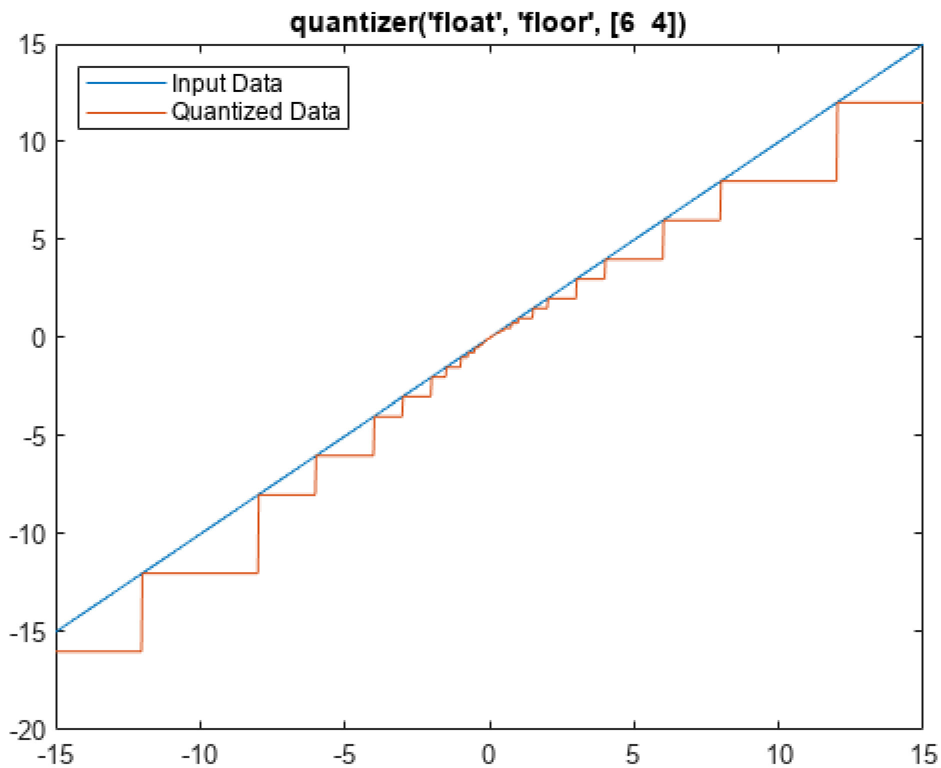
The RoundMode property uses the default setting of 'Floor'.

Use the `quantize` function to quantize the data in `x` using the properties specified by the `quantizer` object.

```
y = quantize(q,x);
```

Plot `y` against `x` to visualize the effect of the specified quantization properties on this data.

```
plot(x,x,x,y); title(tostring(q));
legend('Input Data','Quantized Data','Location','northwest');
```



You can use read-only properties of the `quantizer` object to access more information.

```
q.noverflows
```

```
ans = 0
```

```
q.nunderflows
```

```
ans = 0
```

In this example, there were 0 overflows and 0 underflows that occurred in the quantization operation.

Quantize to Fixed-Point

Create a `quantizer` object specifying a signed fixed-point data mode with a word length of 6 bits, a fraction length of 1 bit, and wrap on overflow.

```
q = quantizer([6 1],'wrap')
```

```
q =

        DataMode = fixed
       RoundMode = floor
    OverflowMode = wrap
          Format = [6  1]
```

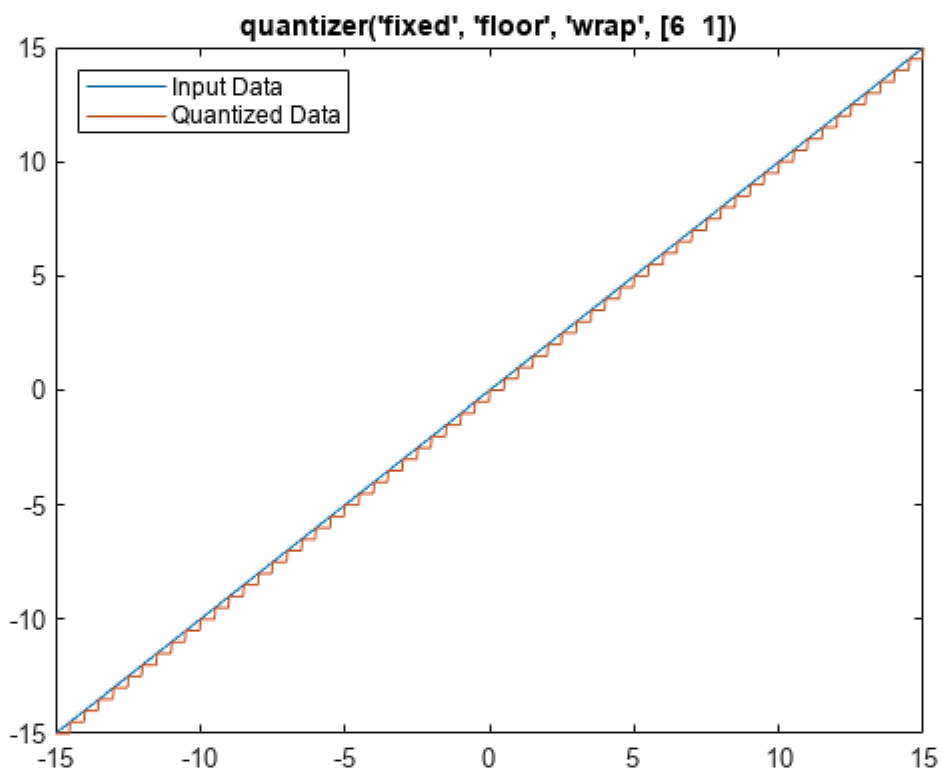
`quantizer` uses the default `DataMode` property, `'fixed'`, and the default `RoundMode` property, `'Floor'`.

Use the `quantize` function to quantize the data in `x` using the properties specified by the `quantizer` object.

```
y = quantize(q,x);
```

Plot `y` against `x` to visualize the effect of the specified quantization properties on this data.

```
plot(x,x,x,y); title(tostring(q));
legend('Input Data','Quantized Data','Location','northwest');
```



You can use read-only properties of the `quantizer` object to access more information.

```
q.noverflows
```

```
ans = 0
```

```
q.nunderflows
```

```
ans = 17
```

In this example, there were 0 overflows and 17 underflows that occurred in the quantization operation.

Version History

Introduced before R2006a

See Also

`quantize` | `reset` | `unitquantize` | `assignmentquantizer`

randquant

Generate uniformly distributed, quantized random number using `quantizer` object

Syntax

```
randquant(q,n)
randquant(q,m,n)
randquant(q,m,n,p,...)
randquant(q,[m,n])
randquant(q,[m,n,p,...])
```

Description

`randquant(q,n)` uses `quantizer` object `q` to generate an `n`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `n`-by-`n` array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,m,n)` uses `quantizer` object `q` to generate an `m`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `m`-by-`n` array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,m,n,p,...)` uses `quantizer` object q to generate an `m`-by-`n`-by-`p`-by ... matrix with random entries whose values cover the range of `q` when `q` is fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the matrix with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,[m,n])` uses `quantizer` object `q` to generate an `m`-by-`n` matrix with random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `m`-by-`n` array with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant(q,[m,n,p,...])` uses `quantizer` object `q` to generate `p` `m`-by-`n` matrices containing random entries whose values cover the range of `q` when `q` is a fixed-point `quantizer` object. When `q` is a floating-point `quantizer` object, `randquant` populates the `m`-by-`n` arrays with values covering the range

```
-[square root of realmax(q)] to [square root of realmax(q)]
```

`randquant` produces pseudorandom numbers. The number sequence `randquant` generates during each call is determined by the state of the generator. Because MATLAB resets the random number generator state at startup, the sequence of random numbers generated by the function remains the same unless you change the state.

`randquant` works like `rng` in most respects.

Examples

```
q = quantizer([4 3]);
rng('default')
randquant(q,3)
```

```
ans =

       0.5     0.625      -0.5
     0.625     0.125         0
    -0.875    -0.875      0.75
```

Version History

Introduced before R2006a

See Also

quantizer | rand | range | realmax

range

Numerical range of fi or quantizer object

Syntax

```
y = range(a)
[min_a,max_a] = range(a)
```

```
r = range(q)
[min_q,max_q] = range(q)
```

Description

Range of fi Object

`y = range(a)` returns a `fi` object with the minimum and maximum possible values of the `fi` object `a`. All possible quantized real-world values of `a` are in the range returned. If `a` is a complex number, then all possible values of `real(a)` and `imag(a)` are in the range returned.

`[min_a,max_a] = range(a)` returns the minimum and maximum values of `fi` object `a` in separate output variables.

Range of quantizer Object

`r = range(q)` returns the two-element row vector `r = [min_q max_q]` such that for all real `x`, `y = quantize(q,x)` returns `y` in the range `min_q ≤ y ≤ max_q`.

`[min_q,max_q] = range(q)` returns the minimum and maximum values of the range in separate output variables.

Examples

Range of `fi` Object

Create a signed `fi` object with a value of 0, word length of 4, and fraction length of 2.

```
a = fi(0,true,4,2);
```

Find the numerical range of the `fi` object `a` and return the result in `fi` object `y`.

```
y = range(a)
```

```
y = 
   -2.0000    1.7500

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 4
        FractionLength: 2
```

Find the numerical range of the `fi` object `a` and return the result in separate output variables.

```
[min_a, max_a] = range(a)
```

```
min_a =
    -2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 4
        FractionLength: 2

max_a =
    1.7500

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 4
        FractionLength: 2
```

Note that `max_a = 1.75 = 2 - eps(a)`.

Range of `quantizer` Object

Create a `quantizer` object that describes a floating-point data type having a word length of 6 and an exponent length of 3. Find the numerical range of the `quantizer` object `q`.

```
q = quantizer('float',[6 3]);
r = range(q)
```

```
r = 1×2

   -14    14
```

Create a `quantizer` object that describes a signed fixed-point data type having a word length of 4, and fraction length of 2, saturate on overflow, and round to floor. Find the numerical range of the `quantizer` object `q` and return the result in separate output variables.

```
q = quantizer('fixed',[4 2],'floor');
[min_q, max_q] = range(q)
```

```
min_q = -2

max_q = 1.7500
```

Note that `max_q = 1.75 = 2 - eps(q)`.

Input Arguments

a — `fi` object
`fi` object

Input `fi` object.

Data Types: `fi`

Complex Number Support: Yes

q — quantizer object
quantizer object

Input quantizer object.

Output Arguments

y — Numerical range of fi object
fi object

Numerical range of input fi object a, returned as a fi object. y is a two-element row vector containing the minimum and maximum possible values of fi object a.

min_a — Minimum value of fi object
fi object

Minimum value of input fi object a, returned as a scalar fi object.

max_a — Maximum value of fi object
fi object

Maximum value of input fi object a, returned as a scalar fi object.

r — Numerical range of quantizer object
two-element row vector

Numerical range of quantizer object q, returned as the two-element row vector r = [min_q max_q] such that for all real x, y = quantize(q,x) returns y in the range min_q ≤ y ≤ max_q.

min_q — Minimum value of quantizer object range
scalar

Minimum value of quantizer object range, returned as a scalar.

max_q — Maximum value of quantizer object range
scalar

Maximum value of quantizer object range, returned as a scalar.

Algorithms

If q is a floating-point quantizer object, min_q = -realmax(q) and max_q = realmax(q).

If q is a signed fixed-point quantizer object (datamode = 'fixed'), then

$$min\_q = -\mathrm{realmax}(q) - \mathrm{eps}(q) = -2^{\mathrm{w}-1}/2^{\mathrm{f}}$$

$$max\_q = \mathrm{realmax}(q) = (2^{\mathrm{w}-1} - 1)/2^{\mathrm{f}}$$

where w is the word length and f is the fraction length.

If q is an unsigned fixed-point quantizer object (datamode = 'ufixed'),

$a = 0$

$b = \text{realmax}(q) = (2^{w} - 1)/2^{f}$

See `realmax` for more information.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eps` | `exponentmax` | `exponentmin` | `fractionlength` | `intmax` | `intmin` | `lowerbound` | `lsb` | `max` | `min` | `realmax` | `realmin` | `upperbound`

rdivide, ./

Package: embedded

Right-array division

Syntax

```
X = A./B
X = rdivide(A,B)
```

Description

X = A./B performs right-array division by dividing each element of A by the corresponding element of B.

X = rdivide(A,B) is an alternative way to execute X = A./B.

Examples

Perform Right-Array Division of Two Matrices

This example shows how perform right-array division on a 3-by-3 magic square of fi objects. Each element of the 3-by-3 magic square is divided by the corresponding element in the 3-by-3 input array b.

The rdivide function outputs a 3-by-3 array of signed fi objects, each of which has a word length of 16 bits and fraction length of 11 bits.

```
a = fi(magic(3))
```

```
a = 
     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
b = int8([3 3 4; 1 2 4 ; 3 1 2 ])
```

```
b = 3x3 int8 matrix

   3   3   4
   1   2   4
   3   1   2
```

```
c = a./b
```

```
c = 
    2.6665    0.3335    1.5000
    3.0000    2.5000    1.7500
    1.3335    9.0000    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Input Arguments

A — Numerator
scalar | vector | matrix | multidimensional array

Numerator, specified as a scalar, vector, matrix, or multidimensional array. Inputs `A` and `B` must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

If `A` is complex, the real and imaginary parts of `A` are independently divided by `B`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

B — Denominator
scalar | vector | matrix | multidimensional array

Denominator, specified as a scalar, vector, matrix, or multidimensional array. Inputs `A` and `B` must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

Output Arguments

X — Quotient
scalar | vector | matrix | multidimensional array

Quotient, returned as a scalar, vector, matrix, or multidimensional array.

The following table shows the rules used to assign property values to the output of the `rdivide` function.

Output Property	Rule
`Signedness`	If either input is `Signed`, then the output is `Signed`. If both inputs are `Unsigned`, then the output is `Unsigned`.
`WordLength`	The output word length equals the maximum of the input word lengths.

Output Property	Rule
`FractionLength`	For `c = a./b`, the fraction length of output `c` equals the fraction length of `a` minus the fraction length of `b`.

Algorithms

The following table shows the rules the `rdivide` function uses to handle inputs with different data types.

Case	Rule
Interoperation of `fi` objects and built-in integers	Built-in integers are treated as fixed-point objects. For example, `B = int8(2)` is treated as an `s8,0` `fi` object.
Interoperation of `fi` objects and constants	MATLAB for code generation treats constant integers as fixed-point objects with the same word length as the `fi` object and a fraction length of `0`.
Interoperation of mixed data types	Similar to all other `fi` object functions, when inputs `a` and `b` have different data types, the data type with the higher precedence determines the output data type. The order of precedence is as follows: **1** `ScaledDouble` **2** `Fixed-point` **3** Built-in `double` **4** Built-in `single` When both inputs are `fi` objects, the only data types that are allowed to mix are `ScaledDouble` and `Fixed-point`.

Version History

Introduced in R2009a

R2022a: Implicit expansion change affects arguments for operators

Behavior changed in R2022a

Starting in R2022a with the addition of implicit expansion for `fi` `rdivide` (`./`), some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`add` | `divide` | `fi` | `fimath` | `mrdivide` | `numerictype` | `sub` | `sum`

realmax

Largest positive fixed-point value or quantized number

Syntax

```
realmax(a)
realmax(q)
```

Description

`realmax(a)` is the largest real-world value that can be represented in the data type of `fi` object `a`. Anything larger overflows.

`realmax(q)` is the largest quantized number that can be represented where `q` is a `quantizer` object. Anything larger overflows.

Examples

```
q = quantizer('float',[6 3]);
x = realmax(q)

x =

    14
```

Algorithms

If `q` is a floating-point `quantizer` object, the largest positive number, x, is

$$x = 2^{E_{max}} \cdot (2 - eps(q))$$

If `q` is a signed fixed-point `quantizer` object, the largest positive number, x, is

$$x = \frac{2^{w-1} - 1}{2^f}$$

If `q` is an unsigned fixed-point `quantizer` object `(datamode = 'ufixed')`, the largest positive number, x, is

$$x = \frac{2^w - 1}{2^f}$$

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`eps` | `exponentmax` | `exponentmin` | `fractionlength` | `intmax` | `intmin` | `lowerbound` | `lsb` | `quantizer` | `range` | `realmin` | `upperbound`

realmin

Smallest positive normalized fixed-point value or quantized number

Syntax

```
x=realmin(a)
x=realmin(q)
```

Description

`x=realmin(a)` is the smallest positive real-world value that can be represented in the data type of `fi` object `a`. Anything smaller than `x` underflows or is an IEEE "denormal" number.

`x=realmin(q)` is the smallest positive normal quantized number where `q` is a `quantizer` object. Anything smaller than `x` underflows or is an IEEE "denormal" number.

Examples

```
q = quantizer('float',[6 3]);
x = realmin(q)

x =

             0.25
```

Algorithms

If `q` is a floating-point `quantizer` object, $x = 2^{E_{min}}$ where E_{min} = exponentmin(q) is the minimum exponent.

If `q` is a signed or unsigned fixed-point `quantizer` object, $x = 2^{-f} = \varepsilon$ where f is the fraction length.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

eps | exponentmax | exponentmin | fractionlength | intmax | intmin | lowerbound | lsb | range | realmax | upperbound

reinterpretcast

Convert fixed-point or integer data types without changing underlying data

Syntax

`c = reinterpretcast(a,T)`

Description

`c = reinterpretcast(a,T)` converts the input `a` to the data type specified by `numerictype` object `T` without changing the underlying data. The result is returned in `fi` object `c`.

The `reinterpretcast` function differs from the MATLAB `typecast` and `cast` functions in that it only operates on `fi` objects and built-in integers, and it does not allow the word length of the input to change.

Examples

Convert `fi` Object to New Data Type

In this example, `a` is a signed `fi` object with a word length of 8 its and a fraction length of 7 bits. The `reinterpretcast` function converts `a` into an unsigned `fi` object `c` with a word length of 8 bits and a fraction length of 0 bits. The real-world values of `a` and `c` are different, but their binary representations are the same.

```
a = fi([-1 pi/4],1,8,7)
```

```
a = 
   -1.0000    0.7891

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

```
T = numerictype(0,8,0);
c = reinterpretcast(a,T)
```

```
c = 
   128   101

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 0
```

To verify that the underlying data has not changed, compare the binary representations of `a` and `c`.

```
binary_a = bin(a)
```

```
binary_a = 
'10000000   01100101'
```

```
binary_c = bin(c)
```

```
binary_c =
'10000000   01100101'
```

Input Arguments

a — Input fixed-point or integer array

scalar | vector | matrix | multidimensional array

Input fixed-point or integer array, specified as a scalar, vector, matrix, or multidimensional array.

The word length of inputs `a` and `T` must be the same.

Data Types: `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

T — New data type

`numerictype` object

New data type, specified as a `numerictype` object that fully specified a fixed-point data type.

The word length of inputs `a` and `T` must be the same.

Version History

Introduced in R2008b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`cast` | `fi` | `numerictype` | `typecast`

removefimath

Remove fimath object from `fi` object

Syntax

```
y = removefimath(x)
```

Description

`y = removefimath(x)` returns a `fi` object `y` with `x`'s `numerictype` and value, and no fimath object attached. You can use this function as `y = removefimath(y)`, which gives you localized control over the fimath settings. This function also is useful for preventing errors about `embedded.fimath` of both operands needing to be equal.

Examples

Remove fimath Object from fi Object

This example shows how to define a `fi` object, define a fimath object, attach the `fimath` object to the `fi` object and then, remove the attached `fimath` object.

```
a = fi(pi)

a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

f = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
a = setfimath(a,f)

a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision

b = removefimath(a)

b = 
    3.1416
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Set and Remove fimath for Code Generation

Use the pattern `x = setfimath(x,f)` and `y = removefimath(y)` to insulate variables from `fimath` settings outside the function. This pattern does not create copies of the data in generated code.

```
function y = fixed_point_32bit_KeepLSB_plus_example(a,b)
   f = fimath('OverflowAction','Wrap',...
      'RoundingMethod','Floor',...
      'SumMode','KeepLSB',...
      'SumWordLength',32);
   a = setfimath(a,f);
   b = setfimath(b,f);
   y = a + b;
   y = removefimath(y);
end
```

If you have the MATLAB Coder product, you can generate C code. This example generates C code on a computer with 32-bit, native integer type.

```
a = fi(0,1,16,15);
b = fi(0,1,16,15);
codegen -config:lib  fixed_point_32bit_KeepLSB_plus_example...
       -args {a,b} -launchreport
```

```
int fixed_point_32bit_KeepLSB_plus_example(short a, short b)
{
  return a + b;
}
```

Input Arguments

x — Input data
fi object | built-in integer | double | single

Input data, specified as a `fi` object or built-in integer, from which to copy the data type and value to the output. `x` must be a `fi` object or an integer data type (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`). If `x` is not a fi object or integer data type, then `y` = `x`.

Output Arguments

y — Output fi object
fi object | built-in integer | double | single

Output fi object, returned as a `fi` object with no fimath object attached. The data type and value of the output match the input. If the input, `x`, is not a `fi` object `y` = `x`.

Version History

Introduced in R2012b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`fi` | `fimath` | `setfimath`

rescale

Change scaling of fi object

Syntax

```
b = rescale(a, fractionlength)
```

```
b = rescale(a, slope, bias)
```

```
b = rescale(a, slopeadjustmentfactor, fixedexponent, bias)
```

```
b = rescale(a, ..., PropertyName, PropertyValue, ...)
```

Description

The `rescale` function acts similarly to the `fi` copy function with the following exceptions:

- The `fi` copy constructor preserves the real-world value, while `rescale` preserves the stored integer value.
- `rescale` does not allow the `Signed` and `WordLength` properties to be changed.

Examples

In the following example, `fi` object `a` is rescaled to create `fi` object `b`. The real-world values of `a` and `b` are different, while their stored integer values are the same:

```
p = fipref('FimathDisplay','none',...
        'NumericTypeDisplay','short');
a = fi(10, 1, 8, 3)

a = 

    10
      numerictype(1,8,3)


b = rescale(a,1)

b = 

    40
      numerictype(1,8,1)

stored_integer_a = storedInteger(a);
stored_integer_b = storedInteger(b);
isequal(stored_integer_a,stored_integer_b)

ans =

  logical

   1
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`fi`

reset

Reset objects to initial conditions

Syntax

```
reset(P)
reset(q)
```

Description

`reset(P)` resets the `fipref` object `P` to its initial conditions.

`reset(q)` resets the following `quantizer` object properties to their initial conditions:

- `minlog`
- `maxlog`
- `noverflows`
- `nunderflows`
- `noperations`

Version History

Introduced before R2006a

See Also

`resetlog`

reset

Clear stored information from NumericTypeScope object

Syntax

```
reset(H)
```

Description

reset(H) clears all stored information from the NumericTypeScope object H. Doing so allows you to reuse H to process data from a different variable. Resetting the object clears the information displayed in the scope window.

Examples

View the Dynamic Range of a fi Object

Use the NumericTypeScope to view the dynamic range of a fi object.

Create a fi object and set the DataTypeOverride to ScaledDoubles.

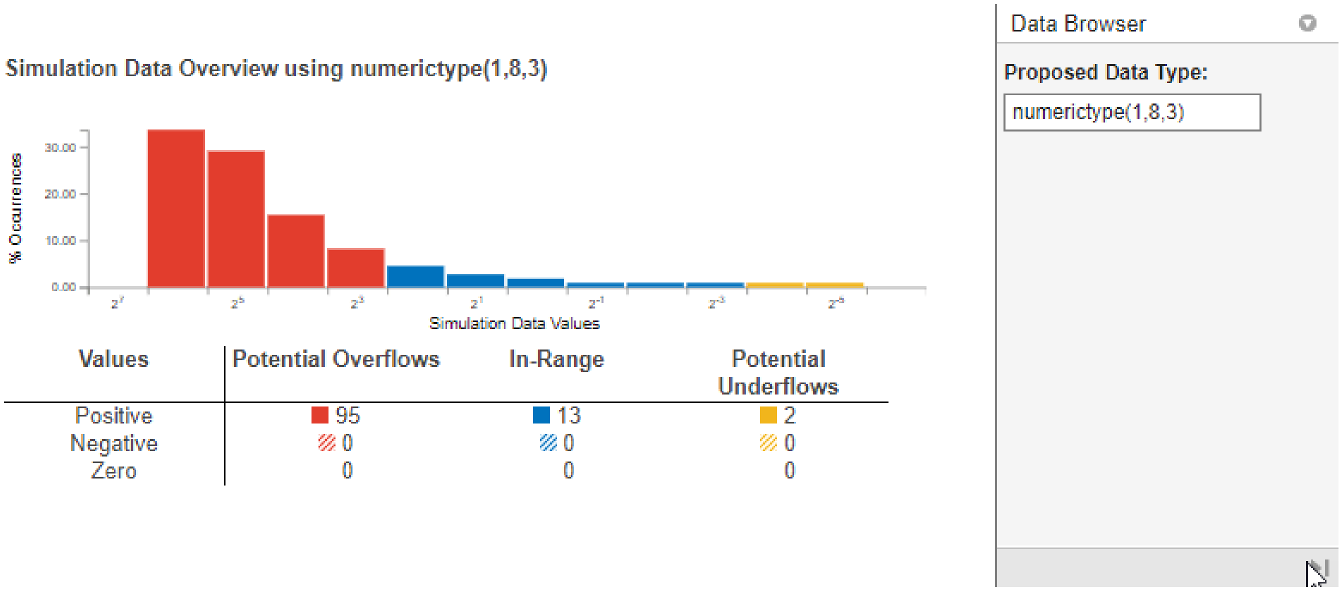
```
a = fi(magic(10),1,8,2);
b = fi([a; 2.^(-5:4)],1,8,3);

fp = fipref;
initialDTOSetting = fp.DataTypeOverride;
fp.DataTypeOverride = 'ScaledDoubles';
```

Create a NumericTypeScope object. You can use the reset method to ensure that all stored information is cleared from the NumericTypeScope object h.

```
h = NumericTypeScope;
reset(h)
```

Use the step method to process your data and visualize the dynamic range of the fi object b.

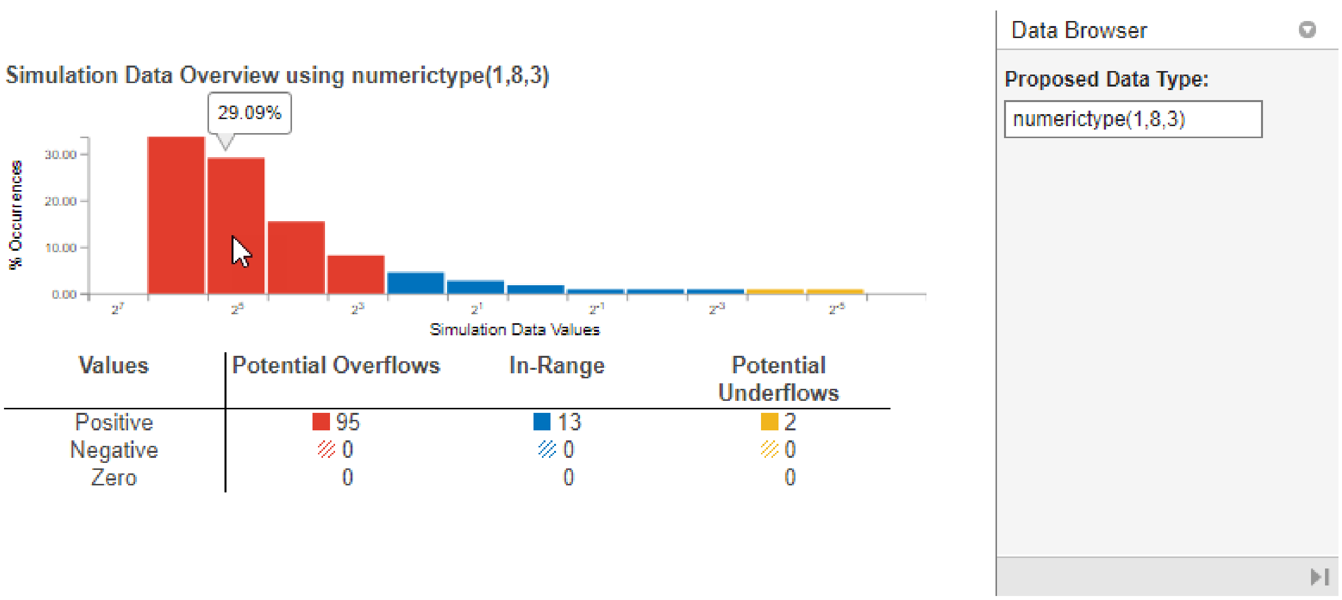
```
step(h,b);
```



Closing the `NumericTypeScope` window does not delete the object from your workspace. Close the `NumericTypeScope` window and reopen it using the `show` function.

```
show(h);
```

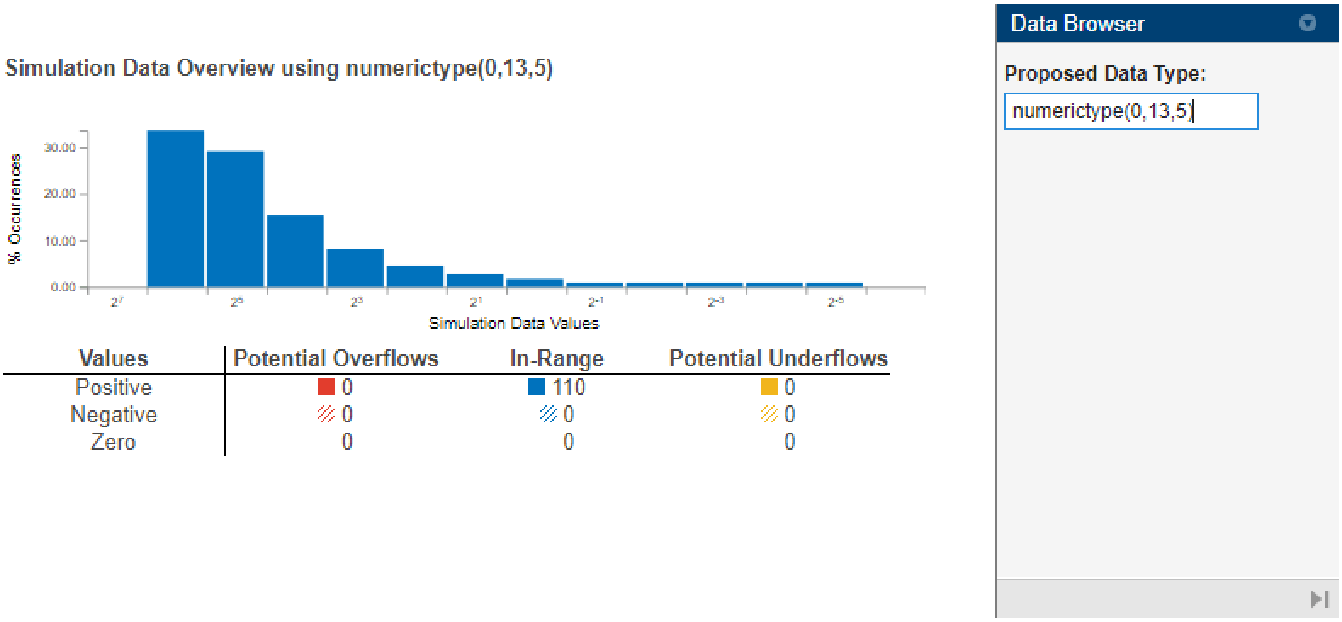
The `NumericTypeScope` displays a `log2` histogram which shows that the values appear both outside of the range and below the precision of the data type of the variable. Pause on one bar of the histogram to view the percentage of the total values that are represented by that bar.



In this case, the data type of `b` is `numerictype(1,8,3)`. The `numerictype(1,8,3)` data type provides 5 integer bits, including the signed bit, and 3 fractional bits. Thus, this data type can represent only values between -2^4 and 2^4 - 2^-3 (from -16 to 15.8750). Given the range and precision of this data type, values greater than 2^4 fall outside the range and values less than 2^-3 fall below the precision of the data type.

The NumericTypeScope shows that values requiring bits 5, 6, and 7 are outside the range and values requiring fractional bits 4 and 5 are below precision. Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable b to numerictype(0,13,5).

Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable b to numerictype(0,13,5). In the NumericTypeScope, enter numerictype(0,13,5) in the **Proposed Data Type** box.



Return to the original data type override setting.

```
fp.DataTypeOverride = initialDTOSetting;
```

Input Arguments

H — NumericTypeScope object

NumericTypeScope object

NumericTypeScope object, specified as a NumericTypeScope object.

Example: reset(H)

Version History

Introduced in R2010a

See Also

NumericTypeScope | step | show

resetglobalfimath

Set global fimath to MATLAB factory default

Syntax

```
resetglobalfimath
```

Description

`resetglobalfimath` sets the global fimath to the MATLAB factory default in your current MATLAB session. The MATLAB factory default has the following properties:

```
      RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
```

Examples

In this example, you create your own `fimath` object `F` and set it as the global fimath. Then, using the `resetglobalfimath` command, reset the global fimath to the MATLAB factory default setting.

```
F = fimath('RoundingMethod','Floor','OverflowAction','Wrap');
globalfimath(F);
F1 = fimath
a = fi(pi)

F1 =


        RoundingMethod: Floor
        OverflowAction: Wrap
           ProductMode: FullPrecision
               SumMode: FullPrecision

a =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Now, set the global fimath back to the factory default setting using `resetglobalfimath`:

```
resetglobalfimath;
F2 = fimath
a = fi(pi)

F2 =
```

```
      RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision
a =

    3.1416

       DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

You've now set the global fimath in your current MATLAB session back to the factory default setting. To use the factory default setting of the global fimath in future MATLAB sessions, you must use the `removeglobalfimathpref` command.

Alternatives

`reset(G)` — If *G* is a handle to the global fimath, `reset(G)` is equivalent to using the `resetglobalfimath` command.

Version History

Introduced in R2010a

See Also

`fimath` | `globalfimath` | `removeglobalfimathpref`

removeglobalfimathpref

Remove global fimath preference

Syntax

`removeglobalfimathpref`

Description

`removeglobalfimathpref` removes your global fimath from the MATLAB preferences. Once you remove the global fimath from your preferences, you cannot save it to them again. It is best practice to remove global fimath from the MATLAB preferences so that you start each MATLAB session using the default `fimath` settings.

The `removeglobalfimathpref` function does not change the global fimath for your current MATLAB session. To revert back to the factory default setting of the global fimath in your current MATLAB session, use the `resetglobalfimath` command.

Examples

Example 4.4. Removing Your Global fimath from the MATLAB Preferences

Typing

```
removeglobalfimathpref;
```

at the MATLAB command line removes your global fimath from the MATLAB preferences. Using the `removeglobalfimathpref` function allows you to:

- Continue using your global fimath in the current MATLAB session
- Use the MATLAB factory default setting of the global fimath in all future MATLAB sessions

To revert back to the MATLAB factory default setting of the global fimath in both your current and future MATLAB sessions, use both the `resetglobalfimath` and the `removeglobalfimathpref` commands:

```
resetglobalfimath;
removeglobalfimath;
```

Version History

Introduced in R2010a

See Also

`fimath` | `globalfimath` | `resetglobalfimath`

resetlog

Clear log for `fi` or `quantizer` object

Syntax

```
resetlog(a)
resetlog(q)
```

Description

`resetlog(a)` clears the log for `fi` object a.

`resetlog(q)` clears the log for `quantizer` object q.

Turn logging on or off by setting the `fipref` property `LoggingMode`.

Version History

Introduced before R2006a

See Also

`fipref` | `maxlog` | `minlog` | `noperations` | `noverflows` | `nunderflows` | `reset`

round

Round fi object toward nearest integer or round input data using quantizer object

Syntax

```
y = round(a)
y = round(q,x)
```

Description

`y = round(a)` rounds fi object a to the nearest integer. In the case of a tie, round rounds values to the nearest integer with greater absolute value. The rounded value is returned in fi object y.

`y = round(q,x)` uses the RoundingMethod and FractionLength settings of quantizer object q to round the numeric data x, but does not check for overflows during the operation. Input x must be a built-in numeric variable. Use the cast function to work with fi objects.

Examples

Use round on a Signed fi Object

The following example demonstrates how the round function affects the numerictype properties of a signed fi object with a word length of 8 and a fraction length of 3.

```
a = fi(pi,1,8,3)
```

```
a = 
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

```
y = round(a)
```

```
y = 
     3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 6
        FractionLength: 0
```

The following example demonstrates how the round function affects the numerictype properties of a signed fi object with a word length of 8 and a fraction length of 12.

```
a = fi(0.025,1,8,12)
```

```
a = 
    0.0249
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 12
```

```
y = round(a)
```

```
y = 
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 2
        FractionLength: 0
```

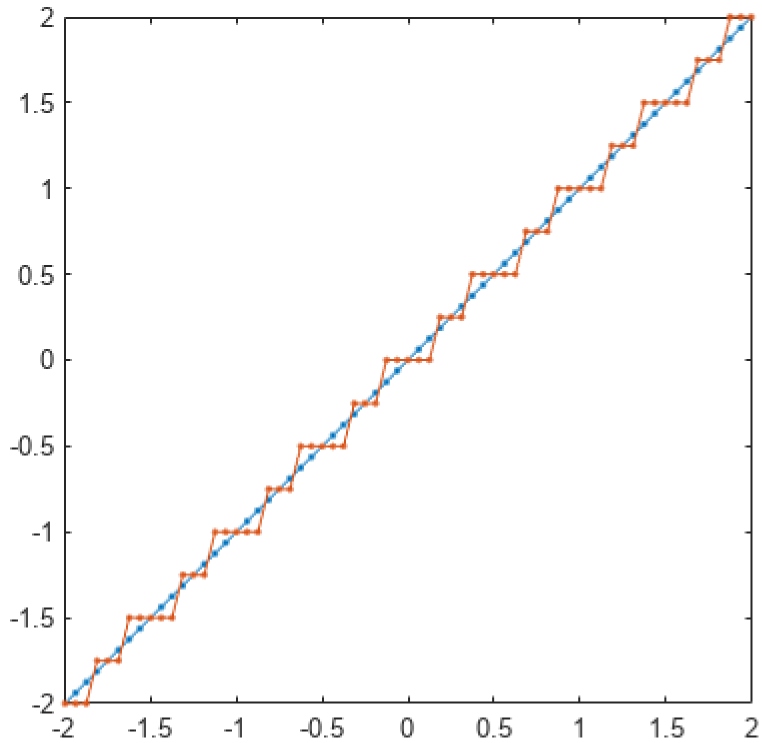
Use `quantizer` Object to Round Numeric Data

This example shows how to use the rounding method and fraction length specified by `quantizer` object `q` to round the numeric data in `x`.

```
q = quantizer('fixed','convergent','wrap',[3 2])
```

```
q = 


        DataMode = fixed
       RoundMode = convergent
    OverflowMode = wrap
          Format = [3  2]
```

```
x = (-2:eps(q)/4:2)';
y = round(q,x);
plot(x,[x,y],'.-'); axis square
```



Compare Rounding Methods

The functions `convergent`, `nearest`, and `round` differ in the way they treat values whose least significant digit is 5.

- The `convergent` function rounds ties to the nearest even integer.
- The `nearest` function rounds ties to the nearest integer toward positive infinity.
- The `round` function rounds ties to the nearest integer with greater absolute value.

This example illustrates these differences for a given input, a.

```
a = fi([-3.5:3.5]');
y = [a convergent(a) nearest(a) round(a)]
```

```
y =
   -3.5000   -4.0000   -3.0000   -4.0000
   -2.5000   -2.0000   -2.0000   -3.0000
   -1.5000   -2.0000   -1.0000   -2.0000
   -0.5000         0         0   -1.0000
    0.5000         0    1.0000    1.0000
    1.5000    2.0000    2.0000    2.0000
    2.5000    2.0000    3.0000    3.0000
    3.5000    3.9999    3.9999    3.9999
```

```
  DataTypeMode: Fixed-point: binary point scaling
    Signedness: Signed
    WordLength: 16
FractionLength: 13
```

Input Arguments

a — Input `fi` array
scalar | vector | matrix | multidimensional array

Input `fi` array, specified as scalar, vector, matrix, or multidimensional array.

For complex `fi` objects, the imaginary and real parts are rounded independently.

`round` does not support `fi` objects with nontrivial slope and bias scaling. Slope and bias scaling is trivial when the slope is an integer power of 2 and the bias is 0.

Data Types: `fi`
Complex Number Support: Yes

q — `RoundingMethod` and `FractionLength` settings
`quantizer` object

`RoundingMethod` and `FractionLength` settings, specified as a `quantizer` object.

Example: q = quantizer('fixed', 'round', [3 2]);

x — Input array
scalar | vector | matrix | multidimensional array

Input array to quantize using the `quantizer` object `q`, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

Algorithms

- `y` and `a` have the same `fimath` object and `DataType` property.
- When the `DataType` property of `a` is `single`, `double`, or `boolean`, the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is zero or negative, `a` is already an integer, and the `numerictype` of `y` is the same as that of `a`.
- When the fraction length of `a` is positive, the fraction length of `y` is `0`, its sign is the same as that of `a`, and its word length is the difference between the word length and the fraction length of `a`, plus one bit. If `a` is signed, then the minimum word length of `y` is `2`. If `a` is unsigned, then the minimum word length of `y` is `1`.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`ceil` | `convergent` | `fix` | `floor` | `nearest` | `quantize` | `quantizer`

rsqrt

Reciprocal square root

Syntax

```
Y = rsqrt(X)
```

Description

`Y = rsqrt(X)` returns the reciprocal square root of each element of the half-precision input array, `X`.

Note This function supports only half-precision inputs.

Examples

Reciprocal Square Root of Matrix Elements

Create a matrix of half-precision values.

```
X = half(magic(3))
```

```
X =

  3x3 half matrix

     8     1     6
     3     5     7
     4     9     2
```

Compute the reciprocal square root of each element of *X*.

```
y = rsqrt(X)
```

```
y =

  3x3 half matrix

    0.3535    1.0000    0.4082
    0.5771    0.4473    0.3779
    0.5000    0.3333    0.7070
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a half-precision numeric scalar, vector, matrix, or multidimensional array

Data Types: `Half`

Version History

Introduced in R2018b

See Also

`half`

savefipref

Save `fi` preferences for next MATLAB session

Syntax

`savefipref`

Description

`savefipref` saves the settings of the current `fipref` object for the next MATLAB session.

Version History

Introduced before R2006a

See Also

`fipref`

sdec

Signed decimal representation of stored integer of `fi` object

Syntax

`sdec(a)`

Description

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

`sdec(a)` returns the stored integer of `fi` object `a` in signed decimal format.

Examples

The code

```
a = fi([-1 1],1,8,7);
sdec(a)
```

returns

```
ans =

    '-128    127'
```

Version History

Introduced before R2006a

See Also

`bin` | `dec` | `hex` | `storedInteger` | `oct`

set

Set or display property values for `quantizer` objects

Syntax

`set(q, PropertyValue1, PropertyValue2,...)`

`set(q,s)`

`set(q,pn,pv)`

```
set(q,'PropertyName1',PropertyValue1,'PropertyName2',
PropertyValue2,...)
```

`q.PropertyName = Value`

`s = set(q)`

Description

`set(q, PropertyValue1, PropertyValue2,...)` sets the properties of `quantizer` object `q`. If two property values conflict, the last value in the list is the one that is set.

`set(q,s)`, where `s` is a structure whose field names are object property names, sets the properties named in each field name with the values contained in the structure.

`set(q,pn,pv)` sets the named properties specified in the cell array of strings `pn` to the corresponding values in the cell array `pv`.

`set(q,'PropertyName1',PropertyValue1,'PropertyName2', PropertyValue2,...)` sets multiple property values with a single statement.

Note You can use property name/property value string pairs, structures, and property name/property value cell array pairs in the same call to `set`.

`q.PropertyName = Value` uses dot notation to set property `PropertyName` to `Value`.

`set(q)` displays the possible values for all properties of `quantizer` object `q`.

`s = set(q)` returns a structure containing the possible values for the properties of `quantizer` object `q`.

Version History

Introduced before R2006a

See Also

`get`

setfimath

Attach `fimath` object to `fi` object

Syntax

`Y = setfimath(X,F)`

Description

`Y = setfimath(X,F)` returns a `fi` object `Y` with `X`'s `numerictype` and value, and attached `fimath` object `F`.

The `Y = setfimath(X,F)` syntax does not modify the input `X`. To modify `X`, use `X = setfimath(X,F)`. This usage gives you more localized control over the `fimath` settings without making a data copy in the generated code.

If you use `setfimath` in an expression, such as `a*setfimath(b,F)`, the `fimath` object is used in the temporary variable, but `b` is not modified.

This function and the related `removefimath` function are useful for preventing errors about the `fimath` of both operands needing to be equal.

Examples

Attach `fimath` Object to `fi` Object

Create a `fi` object.

```
a = fi(pi)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Create a `fimath` object and use `setfimath` to attach it to the `fi` object.

```
f = fimath('OverflowAction','Wrap','RoundingMethod','Floor');
b = setfimath(a,f)
```

```
b = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

```
         RoundingMethod: Floor
         OverflowAction: Wrap
            ProductMode: FullPrecision
                SumMode: FullPrecision
```

Set and Remove fimath for Code Generation

This example shows how to use the pattern `X = setfimath(X,F)` and `Y = removefimath(Y)` to insulate variables from `fimath` settings outside the function. This pattern does not create copies of the data in generated code.

```
type fixed_point_32bit_KeepLSB_plus_example.m
```

```
function y = fixed_point_32bit_KeepLSB_plus_example(a,b)
f = fimath('RoundingMethod', 'Floor', ...
           'OverflowAction', 'Wrap', ...
           'SumMode', 'KeepLSB', ...
           'SumWordLength', 32)

a = setfimath(a,f);
b = setfimath(b,f);

y = a + b;
y = removefimath(y);
end
```

```
a = fi(0,1,16,15);
b = fi(0,1,16,15);
```

You can use MATLAB® Coder™ to generate C code. This example generates C code on a computer with a 32-bit native integer type.

```
codegen -config:lib  fixed_point_32bit_KeepLSB_plus_example...
       -args {a,b} -launchreport
Code generation successful: View report
```

Trace the code in the code generation report.

```
int fixed_point_32bit_KeepLSB_plus_example(short a, short b)
{
  /*    Copyright 2011-2012 The MathWorks, Inc. */
  return a + b;
}
```

Input Arguments

X — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

If `X` is a `fi` object or integer data type, then the `fimath` object is applied. Otherwise, the `fimath` object is not applied and `Y = X`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

F — Fixed-point math settings to attach to the output
`fimath` object

Fixed-point math settings to attach to the output, specified as an existing `fimath` object. If `F` is not a `fimath` object, an error occurs.

Output Arguments

Y — Output `fi` object
`fi` object

Output `fi` object, returned as a `fi` object with the same data type and value as the input `X` and the attached `fimath` object `F`.

If the input `X` is not a `fi` object or integer data type, then `Y = X`.

Version History

Introduced in R2012b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`fi` | `fimath` | `fixed.fimathLike` | `removefimath`

sfi

(Not recommended) Construct signed fixed-point numeric object

Note `sfi` is not recommended. Use `fi` instead.

Syntax

```
a = sfi
a = sfi(v)
a = sfi(v,w)
a = sfi(v,w,f)
a = sfi(v,w,slope,bias)
a = sfi(v,w,slopeadjustmentfactor,fixedexponent,bias)
```

Description

`a = sfi` is the default constructor and returns a signed `fi` object with no value, 16-bit word length, and 15-bit fraction length.

The `fi` object created by the `sfi` constructor function has data properties, `fimath` properties, and `numerictype` properties. These properties are described in detail in fi Object Data Properties, fimath Object Properties and numerictype Object Properties.

The `fi` object created by the `sfi` constructor function has no local `fimath` object. You can attach a `fimath` object to that `fi` object if you do not want to use the default fimath settings. For more information, see “fimath Object Construction”.

`a = sfi(v)` returns a signed fixed-point object with value `v`, 16-bit word length, and best-precision fraction length. Best-precision is when the fraction length is set automatically to accommodate the value `v` for the given word length.

`a = sfi(v,w)` returns a signed fixed-point object with value `v`, word length `w`, and best-precision fraction length.

`a = sfi(v,w,f)` returns a signed fixed-point object with value `v`, word length `w`, and fraction length `f`.

`a = sfi(v,w,slope,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slope`, and `bias`.

`a = sfi(v,w,slopeadjustmentfactor,fixedexponent,bias)` returns a signed fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

Examples

Create a Signed fi Object with Default Values

The default constructor returns a signed `fi` object with no value, 16-bit word length, and 15-bit fraction length.

```
a = sfi
```

```
a =

[]

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Create a Signed fi Object with Default Word Length and Best-Precision Fraction Length

Create a signed `fi` object with the default word length of 16 bits and best-precision fraction length.

```
a = sfi(pi)
```

```
a =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Create a Signed fi Object with Best-Precision Fraction Length

If you omit the argument `f`, the fraction length is set automatically to the best precision possible.

```
a = sfi(pi,8)
```

```
a =

    3.1563

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 5
```

Create a Signed fi Object with Specified Word Length and Fraction Length

Create a signed `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits.

```
a = sfi(pi,8,3)
```

```
a =
```

```
    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 3
```

Default `fimath` properties are associated with `a`. When a `fi` object does not have a local `fimath` object, no `fimath` object properties are displayed in its output. To determine whether a `fi` object has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)
```

```
ans =
     0
```

A returned value of `0` means the `fi` object does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

The value `v` can also be an array.

```
a = sfi((magic(3)/10),16,12)
```

```
a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

Input Arguments

v — Value

scalar | vector | matrix | multi-dimensional array

Value of the signed `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

w — Word length

16 (default) | scalar integer

Word length, in bits, of the signed `fi` object, specified as a scalar integer.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

f — Fraction length

15 (default) | scalar integer

Fraction length, in bits, of the signed `fi` object, specified as a scalar integer. If you do not specify a fraction length, the signed `fi` object automatically uses the fraction length that gives the best precision while avoiding overflow for the specified value and word length.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`slope` — Slope
scalar integer

Slope of the scaling, specified as a scalar integer. The following equation represents the real-world value of a slope bias scaled number.

$$real - worldvalue = (slope \times integer) + bias$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`bias` — Bias
scalar

Bias of the scaling, specified as a scalar. The following equation represents the real-world value of a slope bias scaled number.

$$real - worldvalue = (slope \times integer) + bias$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`slopeadjustmentfactor` — Slope adjustment factor
scalar integer

The slope adjustment factor of a slope bias scaled number. The following equation demonstrates the relationship between the slope, fixed exponent, and slope adjustment factor.

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`fixedexponent` — Fixed exponent
scalar integer

The fixed exponent of a slope bias scaled number. The following equation demonstrates the relationship between the slope, fixed exponent, and slope adjustment factor.

$$slope = slopeadjustmentfactor \times 2^{fixedexponent}$$

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History

Introduced in R2009b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All properties related to data type must be constant for code generation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`fi` | `fimath` | `fipref` | `isfimathlocal` | `numerictype` | `quantizer` | `ufi`

Topics
"View Fixed-Point Data"
"Cast fi Objects"

shiftdata

Shift data to operate on specified dimension

Syntax

`[x,perm,nshifts] = shiftdata(x,dim)`

Description

`[x,perm,nshifts] = shiftdata(x,dim)` shifts data `x` to permute dimension `dim` to the first column using the same permutation as the built-in `filter` function. The vector `perm` returns the permutation vector that is used.

If `dim` is missing or empty, then the first non-singleton dimension is shifted to the first column, and the number of shifts is returned in `nshifts`.

`shiftdata` is meant to be used in tandem with `unshiftdata`, which shifts the data back to its original shape. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples

Example 1

1 Create a 3-x-3 magic square:

```
x = fi(magic(3))
```

```
x =

     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

2 Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

```
x =

     8     3     4
     1     5     9
     6     7     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
```

```
        FractionLength: 11

perm =

     2     1


nshifts =

     []
```

The permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`.

3 Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Example 2

1 Define `x` as a row vector:

```
x = 1:5

x =

     1     2     3     4     5
```

2 Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])

x =

     1
     2
     3
     4
     5


perm =

     []


nshifts =
```

```
     1
```

x is returned as a column vector, along with perm, the permutation vector, and nshifts, the number of shifts.

3 Using unshiftdata, restore x to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =

     1     2     3     4     5
```

Version History

Introduced in R2008a

See Also

unshiftdata

show

Open NumericTypeScope object

Syntax

show(H)

Description

show(H) opens the NumericTypeScope object H and brings it into view. Closing the scope window does not delete the object from your workspace. If the scope object still exists in your workspace, you can open it and bring it back into view using the show function.

Examples

View the Dynamic Range of a fi Object

Use the NumericTypeScope to view the dynamic range of a fi object.

Create a fi object and set the DataTypeOverride to ScaledDoubles.

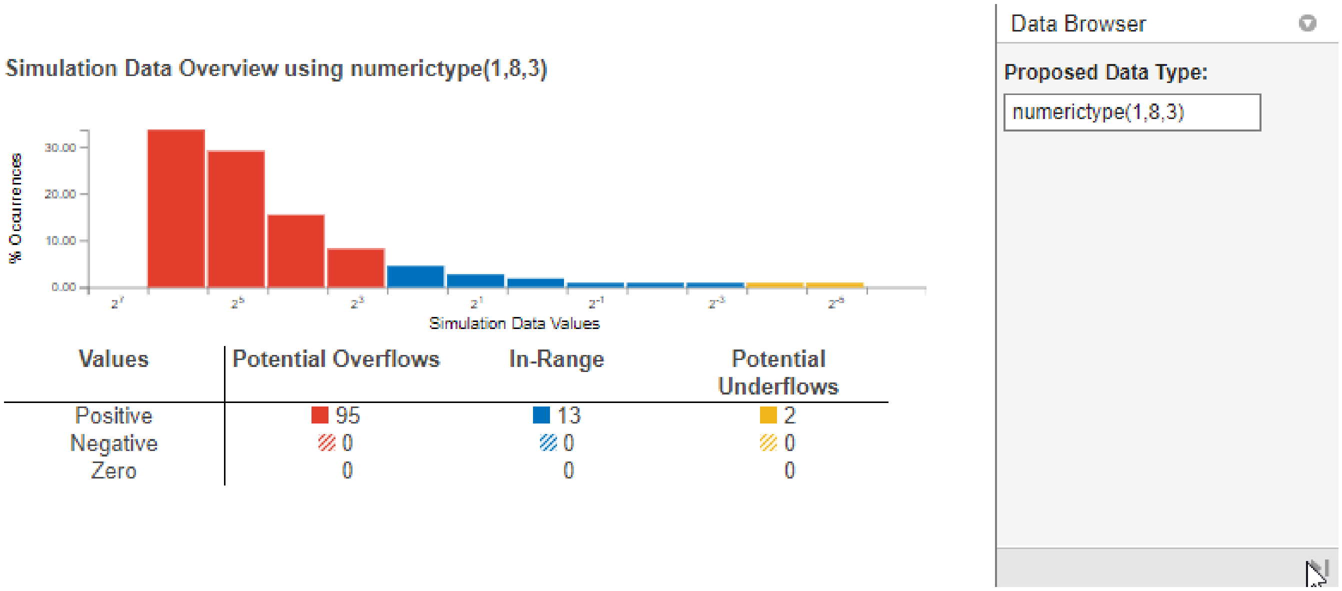
```
a = fi(magic(10),1,8,2);
b = fi([a; 2.^(-5:4)],1,8,3);

fp = fipref;
initialDTOSetting = fp.DataTypeOverride;
fp.DataTypeOverride = 'ScaledDoubles';
```

Create a NumericTypeScope object. You can use the reset method to ensure that all stored information is cleared from the NumericTypeScope object h.

```
h = NumericTypeScope;
reset(h)
```

Use the step method to process your data and visualize the dynamic range of the fi object b.

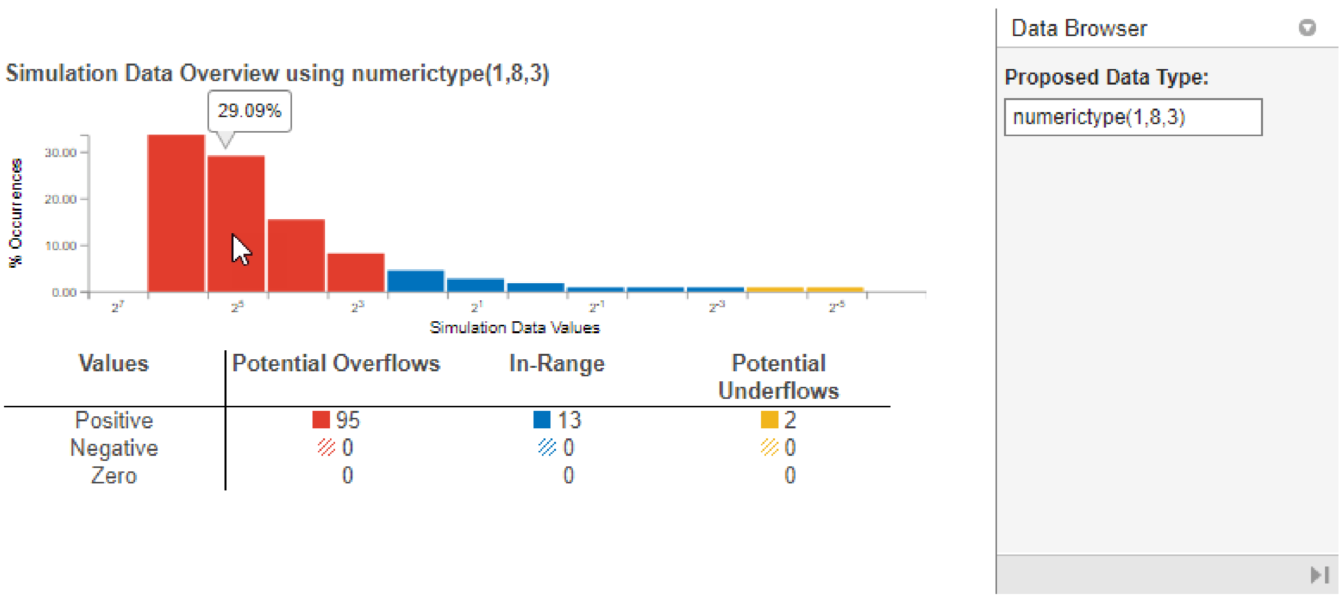
```
step(h,b);
```



Closing the `NumericTypeScope` window does not delete the object from your workspace. Close the `NumericTypeScope` window and reopen it using the `show` function.

```
show(h);
```

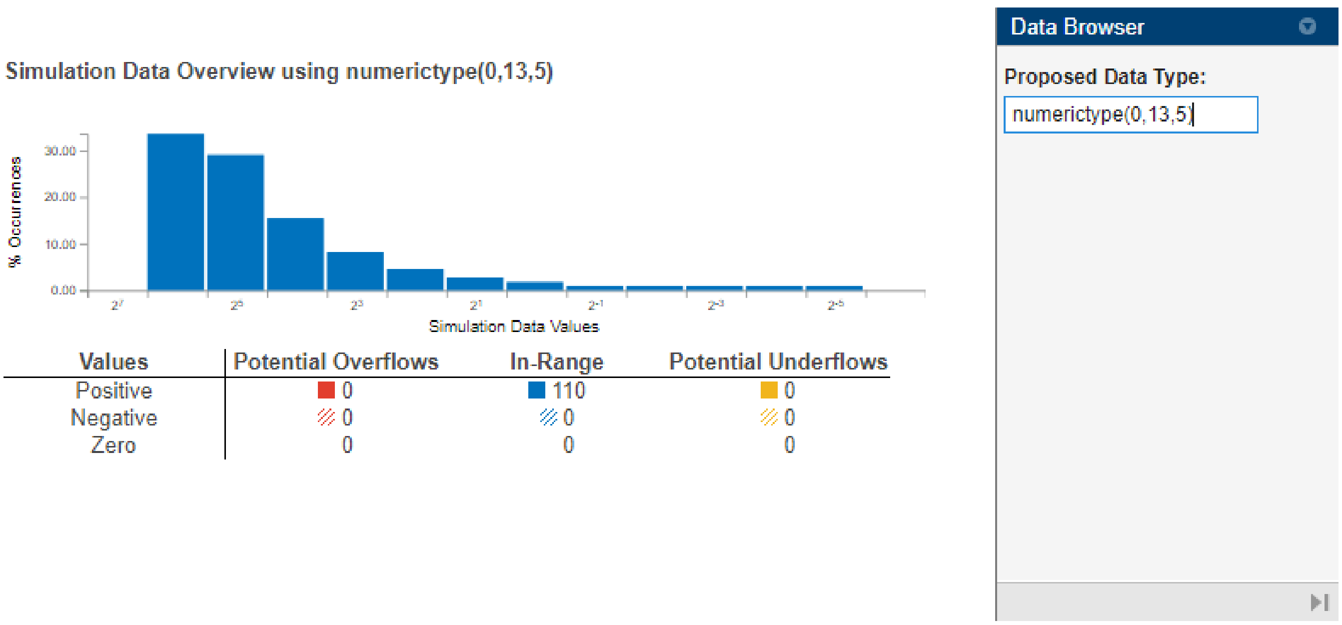
The `NumericTypeScope` displays a `log2` histogram which shows that the values appear both outside of the range and below the precision of the data type of the variable. Pause on one bar of the histogram to view the percentage of the total values that are represented by that bar.



In this case, the data type of `b` is `numerictype(1,8,3)`. The `numerictype(1,8,3)` data type provides 5 integer bits, including the signed bit, and 3 fractional bits. Thus, this data type can represent only values between -2^4 and 2^4 - 2^-3 (from -16 to 15.8750). Given the range and precision of this data type, values greater than 2^4 fall outside the range and values less than 2^-3 fall below the precision of the data type.

The `NumericTypeScope` shows that values requiring bits 5, 6, and 7 are outside the range and values requiring fractional bits 4 and 5 are below precision. Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable `b` to `numerictype(0,13,5)`.

Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable `b` to `numerictype(0,13,5)`. In the `NumericTypeScope`, enter `numerictype(0,13,5)` in the **Proposed Data Type** box.



Return to the original data type override setting.

```
fp.DataTypeOverride = initialDTOSetting;
```

Input Arguments

H — NumericTypeScope object
`NumericTypeScope` object

`NumericTypeScope` object, specified as a `NumericTypeScope` object.

Example: `show(H)`

Version History

Introduced in R2010a

See Also

`NumericTypeScope` | `step` | `reset`

showfixptsimerrors

Show overflows from most recent fixed-point simulation

Compatibility

Note `showfixptsimerrors` will be removed in a future release. Use `fxptdlg` instead.

Syntax

```
showfixptsimerrors
```

Description

The `showfixptsimerrors` script displays any overflows from the most recent fixed-point simulation. This information is also visible in the Fixed-Point Tool.

Version History

Introduced before R2006a

See Also

`autofixexp` | `fxptdlg`

showfixptsimranges

Show logged maximum values, minimum values, and overflow data from fixed-point simulation

Compatibility

Note `showfixptsimranges` will be removed in a future release. Use `fxptdlg` instead.

Syntax

```
showfixptsimranges
showfixptsimranges(action)
```

Description

`showfixptsimranges` displays the logged maximum values, minimum values, and overflow data from the most recent fixed-point simulation in the MATLAB Command Window.

`showfixptsimranges(action)` stores the logged maximum values, minimum values, and overflow data from the most recent fixed-point simulation in the workspace variable `FixPtSimRanges`. If `action` is `'verbose'`, the logged data also appears in the MATLAB Command Window. If `action` is `'quiet'`, no data appears.

Version History

Introduced before R2006a

See Also

`autofixexp` | `fxptdlg`

showInstrumentationResults

Results logged by instrumented, compiled C code function

Syntax

```
showInstrumentationResults('mex_fcn')
showInstrumentationResults('mex_fcn','-options')
showInstrumentationResults mex_fcn
showInstrumentationResults mex_fcn -options
```

Description

`showInstrumentationResults('mex_fcn')` opens the **Instrumentation Report Viewer**, showing results from calling the instrumented MEX function `mex_fcn`.

Hovering over variables and expressions in the report displays the logged information. The logged information includes minimum and maximum values, proposed fraction or word lengths, percent of current range, and whether the value is always a whole number, depending on which options you specify. If you specify to include them in the `buildInstrumentedMex` function, histograms are also included. The same information is displayed in a summary table in the **Variables** tab.

When you call `showInstrumentationResults`, a file named `instrumentation/mex_fcn/html/printable.html` is created. `mex_fcn` is the name of the corresponding instrumented MEX function. Selecting this file opens a web-based version of the **Instrumentation Report Viewer**. To open this file from within MATLAB, right-click on the file and select **Open Outside MATLAB**.

The `showInstrumentationResults` function returns an error if the instrumented `mex_fcn` has not yet been called.

Note The logged results from the `showInstrumentationResults` function are an accumulation of all previous calls to the instrumented MEX function. To clear the log, see `clearInstrumentationResults`.

`showInstrumentationResults('mex_fcn','-options')` specifies options for the instrumentation results section of the **Instrumentation Report Viewer**.

`showInstrumentationResults mex_fcn` and `showInstrumentationResults mex_fcn -options` are alternative syntaxes for opening the **Instrumentation Report Viewer**.

Examples

Create Instrumented MEX Function

This example shows how to create an instrumented MEX function, run a test bench, then view logged results.

Define prototype input arguments.

```
n = 128;
x = complex(zeros(n,1));
w = fi_radix2twiddles(n);
```

Generate an instrumented MEX function. Use the `-o` option to specify the MEX function name. Use the `-histogram` option to compute histograms.

If you have a MATLAB® Coder™ license, you can also add the `-coder` option. In this case,

```
buildInstrumentedMex testfft -o testfft_instrumented -args {x,coder.Constant(w)} -histogram
```

If you have a MATLAB® Coder™ license, you can also add the `-coder` option. For example,

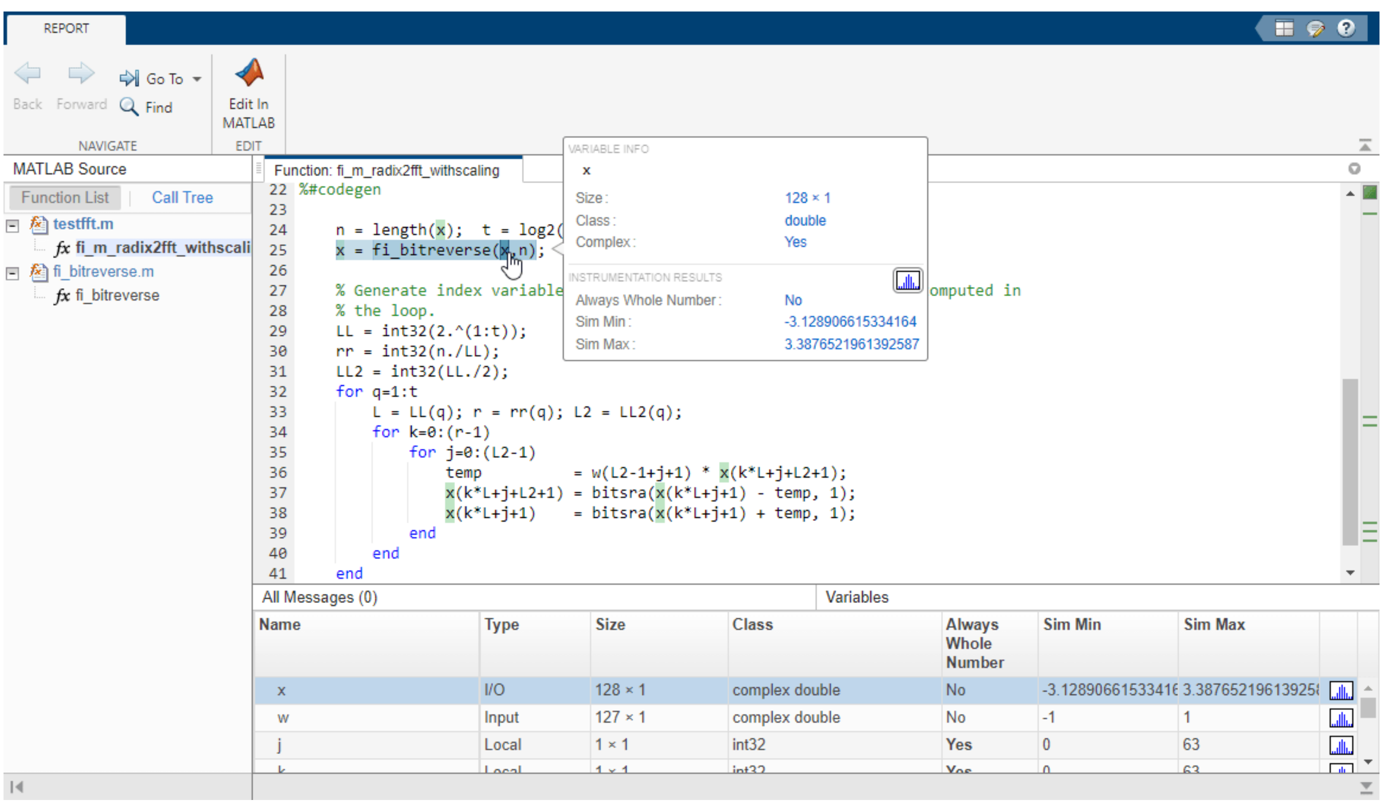
```
buildInstrumentedMex testfft -coder -o testfft_instrumented -args {x,w}
```

Like the `fiaccel` function, the `buildInstrumentedMex` function generates a MEX function. To generate C code, use the MATLAB® Coder™ `codegen` function.

Run a test file to record instrumentation results. Use the `showInstrumentedMex` function to open the report. To view the simulation minimum and maximum values and whole number status, pause over a variable in the report. You can also see proposed data types for double precision numbers in the table.

```
for i=1:20
   y = testfft_instrumented(randn(size(x)),w);
end
```

```
showInstrumentationResults testfft_instrumented
```

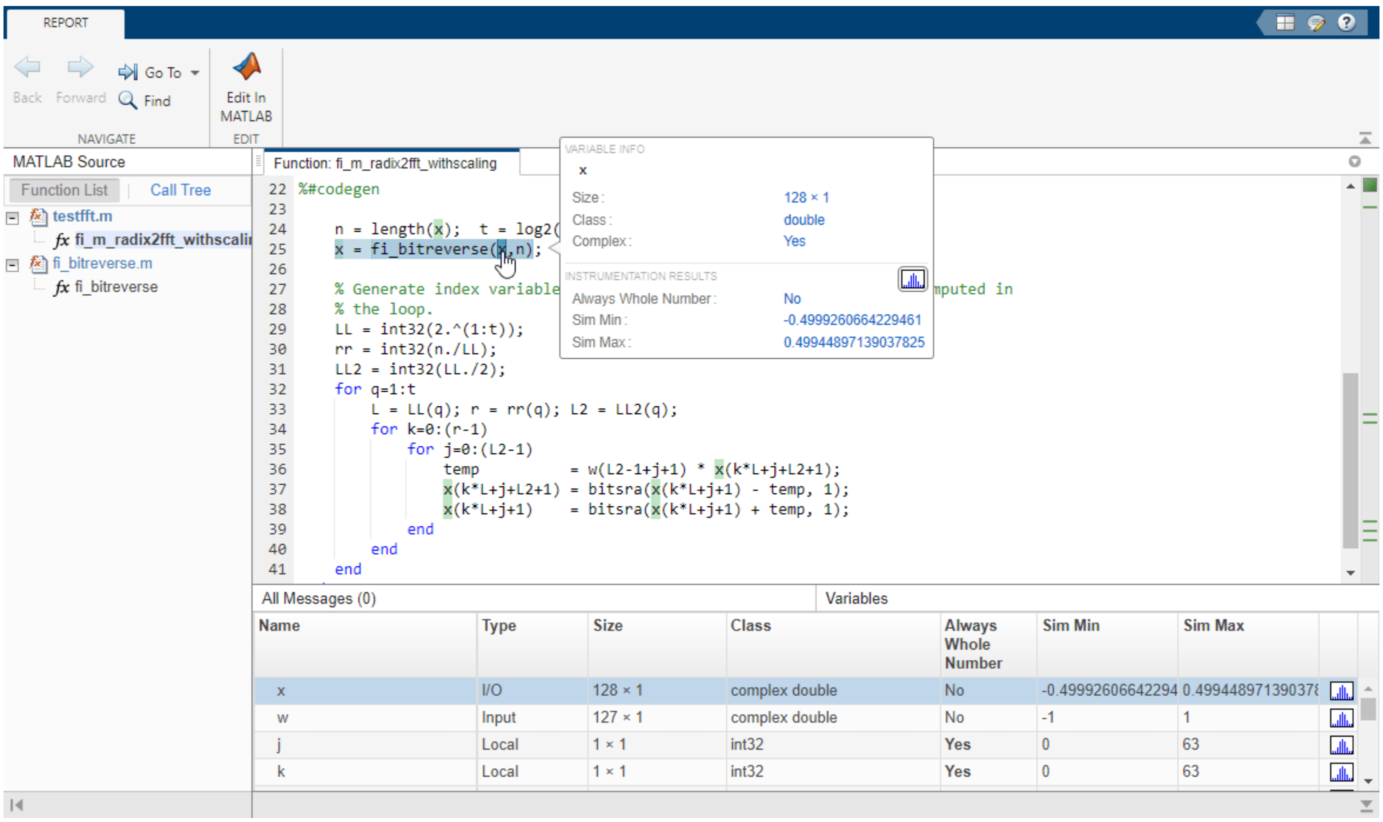


Close the histogram display, then use the `clearInstrumentationResults` function to clear the results log.

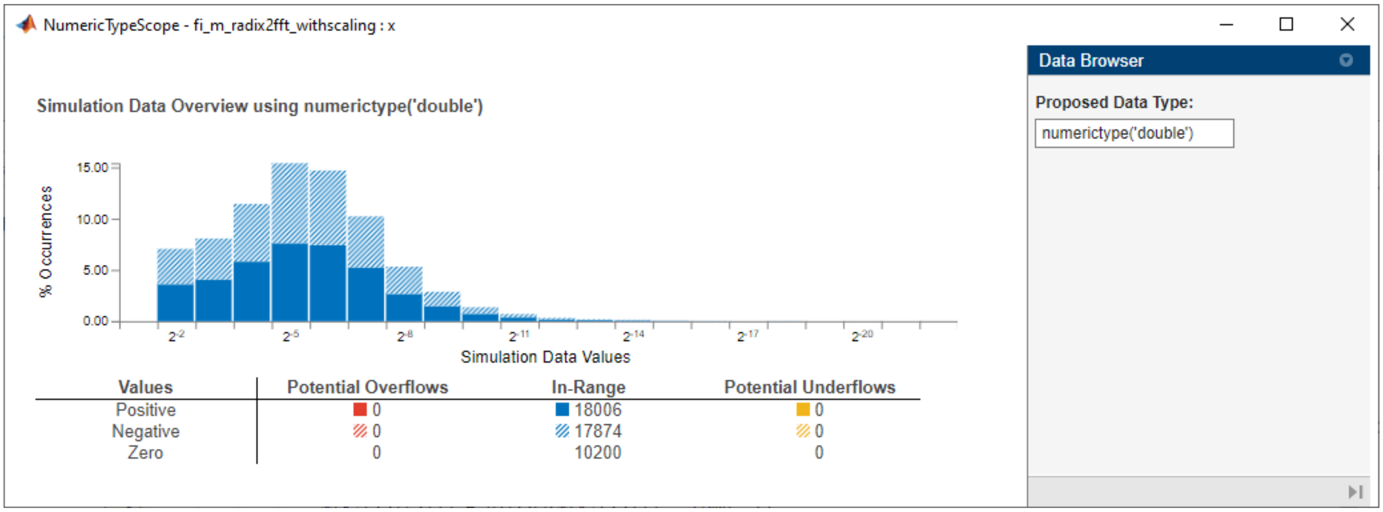
```
clearInstrumentationResults testfft_instrumented
```

Run a different test bench, then view the new instrumentation results.

```
for i=1:20
   y = testfft_instrumented(cast(rand(size(x))-0.5,'like',x),w);
end

showInstrumentationResults testfft_instrumented
```



To view the histogram for a variable, click the histogram icon in the **Variables** tab.



Close the histogram display, then use the `clearInstrumentationResults` function to clear the results log.

```
clearInstrumentationResults testfft_instrumented
```

Clear the MEX function.

```
clear testfft_instrumented
```

Input Arguments

`mex_fcn` — Instrumented MEX function

MEX function

Instrumented MEX function created using `buildInstrumentedMex`.

`options` — Instrumentation results options

`-defaultDT` *`T`* | `-nocode` | `-optimizeWholeNumbers` | `-percentSafetyMargin` *`N`* | `-printable` | `-proposeFL` | `-proposeWL`

Instrumentation results options, specified as:

`-defaultDT` *T*	Default data type to propose for double or single data type inputs, where *T* is either a `numerictype` object or one of the following: `'remainFloat'` (default), `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'int64'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'uint64'`. If you specify an `int` or `uint`, the signedness and word length are that `int` or `uint` value and a fraction length is proposed. The default is `remainFloat`, which does not propose any data types.
`-nocode`	Do not display MATLAB code in the printable report. Display only the tables of logged variables. This option only has effect in combination with the `-printable` option.
`-optimizeWholeNumbers`	Optimize the word length of variables whose simulation min/max logs indicate that they are always whole numbers.
`-percentSafetyMargin` *N*	Safety margin for simulation min/max, where *N* is a percent value.
`-printable`	Create and open a printable HTML report. The report opens in the system browser.
`-proposeFL`	Propose fraction lengths for specified word lengths.
`-proposeWL`	Propose word lengths for specified fraction lengths.

Version History

Introduced in R2011b

R2018a: Redesigned code generation reports

Behavior changed in R2018a

The code generation report `showInstrumentationResults` has a new user interface.

Some benefits of the new interface are:

- Improved navigation. For example, if you double-click a variable in the MATLAB code, you see the variable in the **Variables** tab.
- Easier to use pop-up displays data type information in the `showInstrumentationResults` report. For example, you can pin the pop-up display to the report.

In R2018a, the reports are located in the same folders as in previous releases, but have a different file format. In previous releases, a report was saved with an HTML format and consisted of many files. In R2018a, a report is saved as one file with an `.mldatx` file extension. You can open a file with an `.mldatx` extension in MATLAB.

If you generate a report in R2018a, you cannot open it in a previous release. In R2018a, you can open reports that you generated in a previous release, but they look and behave as they did in that release.

See Also

`fiaccel` | `clearInstrumentationResults` | `buildInstrumentedMex` | `codegen` | `mex`

Simulink.sdi.compareRuns

Package: Simulink.sdi

Compare data in two simulation runs

Syntax

```
diffResult = Simulink.sdi.compareRuns(runID1,runID2)
diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)
```

Description

`diffResult = Simulink.sdi.compareRuns(runID1,runID2)` compares the data in the runs that correspond to `runID1` and `runID2` and returns the result in the `Simulink.sdi.DiffRunResult` object `diffResult`. For more information about the comparison algorithm, see “How the Simulation Data Inspector Compares Data”.

`diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name=Value)` compares the simulation runs that correspond to `runID1` and `runID2` using the options specified by one or more name-value arguments. For more information about comparison options, see “How the Simulation Data Inspector Compares Data”.

Examples

Compare Runs with Global Tolerance

You can specify global tolerance values to use when comparing two simulation runs. Global tolerance values are applied to all signals within the run. This example shows how to specify global tolerance values for a run comparison and how to analyze and save the comparison results.

First, load the session file that contains the data to compare. The session file contains data for four simulations of an aircraft longitudinal controller. This example compares data from two runs that use different input filter time constants.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

To access the run data to compare, use the `Simulink.sdi.getAllRunIDs` function to get the run IDs that correspond to the last two simulation runs.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);
```

Use the `Simulink.sdi.compareRuns` function to compare the runs. Specify a global relative tolerance value of `0.2` and a global time tolerance value of `0.5`.

```
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);
```

Check the `Summary` property of the returned `Simulink.sdi.DiffRunResult` object to see whether signals were within the tolerance values or out of tolerance.

```
runResult.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 3
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

All three signal comparison results fell within the specified global tolerance.

You can save the comparison results to an MLDATX file using the `saveResult` function.

```
saveResult(runResult,'InputFilterComparison');
```

Analyze Simulation Data Using Signal Tolerances

You can programmatically specify signal tolerance values to use in comparisons performed using the Simulation Data Inspector. In this example, you compare data collected by simulating a model of an aircraft longitudinal flight control system. Each simulation uses a different value for the input filter time constant and logs the input and output signals. You analyze the effect of the time constant change by comparing results using the Simulation Data Inspector and signal tolerances.

First, load the session file that contains the simulation data.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains four runs. In this example, you compare data from the first two runs in the file. Access the `Simulink.sdi.Run` objects for the first two runs loaded from the file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Now, compare the two runs without specifying any tolerances.

```
noTolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
```

Use the `getResultByIndex` function to access the comparison results for the `q` and `alpha` signals.

```
qResult = getResultByIndex(noTolDiffResult,1);
alphaResult = getResultByIndex(noTolDiffResult,2);
```

Check the `Status` of each signal result to see whether the comparison result fell within our out of tolerance.

```
qResult.Status
```

```
ans = 
  ComparisonSignalStatus enumeration
```

```
    OutOfTolerance
```

```
alphaResult.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    OutOfTolerance
```

The comparison used a value of `0` for all tolerances, so the `OutOfTolerance` result means the signals are not identical.

You can further analyze the effect of the time constant by specifying tolerance values for the signals. Specify the tolerances by setting the properties for the `Simulink.sdi.Signal` objects that correspond to the signals being compared. Comparisons use tolerances specified for the baseline signals. This example specifies a time tolerance and an absolute tolerance.

To specify a tolerance, first access the `Signal` objects from the baseline run.

```
runTs1 = Simulink.sdi.getRun(runIDTs1);
qSig = getSignalsByName(runTs1,'q, rad/sec');
alphaSig = getSignalsByName(runTs1,'alpha, rad');
```

Specify an absolute tolerance of `0.1` and a time tolerance of `0.6` for the `q` signal using the `AbsTol` and `TimeTol` properties.

```
qSig.AbsTol = 0.1;
qSig.TimeTol = 0.6;
```

Specify an absolute tolerance of `0.2` and a time tolerance of `0.8` for the `alpha` signal.

```
alphaSig.AbsTol = 0.2;
alphaSig.TimeTol = 0.8;
```

Compare the results again. Access the results from the comparison and check the `Status` property for each signal.

```
tolDiffResult = Simulink.sdi.compareRuns(runIDTs1,runIDTs2);
qResult2 = getResultByIndex(tolDiffResult,1);
alphaResult2 = getResultByIndex(tolDiffResult,2);
```

```
qResult2.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    WithinTolerance
```

```
alphaResult2.Status
```

```
ans = 
  ComparisonSignalStatus enumeration

    WithinTolerance
```

Configure Comparisons to Check Metadata

You can use the Simulink.sdi.compareRuns function to compare signal data and metadata, including data type and start and stop times. A single comparison may check for mismatches in one or more pieces of metadata. When you check for mismatches in signal metadata, the Summary property of the Simulink.sdi.DiffRunResult object may differ from a basic comparison because the Status property for a Simulink.sdi.DiffSignalResult object can indicate the metadata mismatch. You can configure comparisons using the Simulink.sdi.compareRuns function for imported data and for data logged from a simulation.

This example configures a comparison of runs created from workspace data three ways to show how the Summary of the DiffSignalResult object can provide specific information about signal mismatches.

Create Workspace Data

The Simulink.sdi.compareRuns function compares time series data. Create data for a sine wave to use as the baseline signal, using the timeseries format. Give the timeseries the name Wave Data.

```
time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';
```

Create a second sine wave to compare against the baseline signal. Use a slightly different time vector and attenuate the signal so the two signals are not identical. Cast the signal data to the single data type. Also name this timeseries object Wave Data. The Simulation Data Inspector comparison algorithm will align these signals for comparison using the name.

```
time2 = 0:0.1:22;
sig2vals = single(0.98*sin(2*pi/5*time2));
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';
```

Create and Compare Runs in the Simulation Data Inspector

The Simulink.sdi.compareRuns function compares data contained in Simulink.sdi.Run objects. Use the Simulink.sdi.createRun function to create runs in the Simulation Data Inspector for the data. The Simulink.sdi.createRun function returns the run ID for each created run.

```
runID1 = Simulink.sdi.createRun('Baseline Run','vars',sig1_ts);
runID2 = Simulink.sdi.createRun('Compare to Run','vars',sig2_ts);
```

You can use the Simulink.sdi.compareRuns function to compare the runs. The comparison algorithm converts the signal data to the double data type and synchronizes the signal data before computing the difference signal.

```
basic_DRR = Simulink.sdi.compareRuns(runID1,runID2);
```

Check the Summary property of the returned Simulink.sdi.DiffRunResult object to see the result of the comparison.

```
basic_DRR.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 1
      WithinTolerance: 0
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 0
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

The difference between the signals is out of tolerance.

Compare Runs and Check for Data Type Match

Depending on your system requirements, you may want the data types for signals you compare to match. You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check for and report data type mismatches.

```
dataType_DRR = Simulink.sdi.compareRuns(runID1,runID2,'DataType','MustMatch');
dataType_DRR.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 0
            Unaligned: 0
        UnitsMismatch: 0
                Empty: 0
             Canceled: 0
          EmptySynced: 0
     DataTypeMismatch: 1
         TimeMismatch: 0
    StartStopMismatch: 0
          Unsupported: 0
```

The result of the signal comparison is now `DataTypeMismatch` because the data for the baseline signal is `double` data type, while the data for the signal compared to the baseline is `single` data type.

Compare Runs and Check for Start and Stop Time Match

You can use the `Simulink.sdi.compareRuns` function to configure the comparison algorithm to check whether the aligned signals have the same start and stop times.

```
startStop_DRR = Simulink.sdi.compareRuns(runID1,runID2,'StartStop','MustMatch');
startStop_DRR.Summary
```

```
ans = struct with fields:
       OutOfTolerance: 0
      WithinTolerance: 0
            Unaligned: 0
        UnitsMismatch: 0
```

```
             Empty: 0
          Canceled: 0
       EmptySynced: 0
  DataTypeMismatch: 0
      TimeMismatch: 0
 StartStopMismatch: 1
       Unsupported: 0
```

The signal comparison result is now `StartStopMismatch` because the signals created in the workspace have different stop times.

Compare Runs with Alignment Criteria

When you compare runs using the Simulation Data Inspector, you can specify alignment criteria that determine how signals are paired with each other for comparison. This example compares data from simulations of a model of an aircraft longitudinal control system. The simulations used a square wave input. The first simulation used an input filter time constant of `0.1s` and the second simulation used an input filter time constant of `0.5s`.

First, load the simulation data from the session file that contains the data for this example.

```
Simulink.sdi.load('AircraftExample.mldatx');
```

The session file contains data for four simulations. This example compares data from the first two runs. Access the run IDs for the first two runs loaded from the session file.

```
runIDs = Simulink.sdi.getAllRunIDs;
runIDTs1 = runIDs(end-3);
runIDTs2 = runIDs(end-2);
```

Before running the comparison, define how you want the Simulation Data Inspector to align the signals between the runs. This example aligns signals by their name, then by their block path, and then by their Simulink identifier.

```
alignMethods = [Simulink.sdi.AlignType.SignalName
               Simulink.sdi.AlignType.BlockPath
               Simulink.sdi.AlignType.SID];
```

Compare the simulation data in your two runs, using the alignment criteria you specified. The comparison uses a small time tolerance to account for the effect of differences in the step size used by the solver on the transition of the square wave input.

```
diffResults = Simulink.sdi.compareRuns(runIDTs1,runIDTs2,'align',alignMethods,...
    'timetol',0.005);
```

You can use the `getResultByIndex` function to access the comparison results for the aligned signals in the runs you compared. You can use the `Count` property of the `Simulink.sdi.DiffRunResult` object to set up a `for` loop to check the `Status` property for each `Simulink.sdi.DiffSignalResult` object.

```
numComparisons = diffResults.count;

for k = 1:numComparisons
    resultAtIdx = getResultByIndex(diffResults,k);
```

```
    sigID1 = resultAtIdx.signalID1;
    sigID2 = resultAtIdx.signalID2;

    sig1 = Simulink.sdi.getSignal(sigID1);
    sig2 = Simulink.sdi.getSignal(sigID2);

    displayStr = 'Signals %s and %s: %s \n';
    fprintf(displayStr,sig1.Name,sig2.Name,resultAtIdx.Status);
end
```

```
Signals q, rad/sec and q, rad/sec: OutOfTolerance
Signals alpha, rad and alpha, rad: OutOfTolerance
Signals Stick and Stick: WithinTolerance
```

Input Arguments

runID1 — Baseline run identifier
integer

Numeric identifier for the baseline run in the comparison, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

runID2 — Identifier for run to compare
integer

Numeric identifier for the run to compare, specified as a run ID that corresponds to a run in the Simulation Data Inspector. The Simulation Data Inspector assigns run IDs when runs are created. You can get the run ID for a run by using the `ID` property of the `Simulink.sdi.Run` object, the `Simulink.sdi.getAllRunIDs` function, or the `Simulink.sdi.getRunIDByIndex` function.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `AbsTol=x,Align=alignOpts`

Align — Signal alignment options
`Simulink.sdi.AlignType` scalar | `Simulink.sdi.AlignType` vector

Signal alignment options, specified as a `Simulink.sdi.AlignType` scalar or vector. The `Simulink.sdi.AlignType` enumeration includes a value for each option available for pairing each signal in the baseline run with a signal in the comparison run. You can specify one or more alignment options for the comparison. To use more than one alignment option, specify an array. When you specify multiple alignment options, the Simulation Data Inspector aligns signals first by the option in the first element of the array, then by the option in the second element array, and so on. For more information, see “Signal Alignment”.

Value	Aligns By
`Simulink.sdi.AlignType.BlockPath`	Path to the source block for the signal
`Simulink.sdi.AlignType.SID`	Automatically assigned Simulink identifier
`Simulink.sdi.AlignType.SignalName`	Signal name
`Simulink.sdi.AlignType.DataSource`	Path of the variable in the MATLAB workspace

Example: `[Simulink.sdi.AlignType.SignalName,Simulink.sdi.AlignType.BlockPath]` specifies signal alignment by signal name and then by block path.

`AbsTol` — Global absolute tolerance for comparison
`0` (default) | positive-valued scalar

Global absolute tolerance for comparison, specified as a positive-valued scalar.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see “Tolerance Specification”.

Example: `0.5`

Data Types: `double`

`RelTol` — Global relative tolerance for comparison
`0` (default) | positive-valued scalar

Global relative tolerance for comparison, specified as a positive-valued scalar. The relative tolerance is expressed as a fractional multiplier. For example, `0.1` specifies a 10 percent tolerance.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about how tolerances are used in comparisons, see “Tolerance Specification”.

Example: `0.1`

Data Types: `double`

`TimeTol` — Global time tolerance for comparison
`0` (default) | positive-valued scalar

Global time tolerance for comparison, specified as a positive-valued scalar, using units of seconds.

Global tolerances apply to all signals in the run comparison. To use a different tolerance value for a signal in the comparison, specify the tolerance you want to use on the `Simulink.sdi.Signal` object in the baseline run and set the `OverrideGlobalTol` property for that signal to `true`.

For more information about tolerances in the Simulation Data Inspector, see “Tolerance Specification”.

Example: `0.2`

Data Types: `double`

DataType — Comparison sensitivity to signal data types
"MustMatch"

Comparison sensitivity to signal data types, specified as "MustMatch". Specify DataType="MustMatch" when you want the comparison to be sensitive to numeric data type mismatches in compared signals.

When signal data types do not match, the Status property of the Simulink.sdi.DiffSignalResult object for the result is set to DataTypeMismatch.

The Simulink.sdi.compareRuns function compares the data types for aligned signals before synchronizing and comparing the signal data. When you do not specify this name-value argument, the comparison checks data types only to detect a comparison between string and numeric data. For a comparison between string and numeric data, results are not computed, and the status for the result is DataTypeMismatch. For aligned signals that have different numeric data types, the comparison computes results.

When you configure the comparison to stop on the first mismatch, a data type mismatch stops the comparison. A stopped comparison may not compute results for all signals.

Time — Comparison sensitivity to signal time vectors
"MustMatch"

Comparison sensitivity to signal time vectors, specified as "MustMatch". Specify Time="MustMatch" when you want the comparison to be sensitive to mismatches in the time vectors of compared signals. When you specify this name-value argument, the algorithm compares the time vectors of aligned signals before synchronizing and comparing the signal data.

When the time vectors for signals do not match, the Status property of the Simulink.sdi.DiffSignalResult object for the result is set to TimeMismatch.

Comparisons are not sensitive to differences in signal time vectors unless you specify this name-value argument. For comparisons that are not sensitive to differences in the time vectors, the comparison algorithm synchronizes the signals prior to the comparison. For more information about how synchronization works, see “How the Simulation Data Inspector Compares Data”.

When you specify that time vectors must match and configure the comparison to stop on the first mismatch, a time vector mismatch stops the comparison. A stopped comparison may not compute results for all signals.

StartStop — Comparison sensitivity to signal start and stop times
"MustMatch"

Comparison sensitivity to signal start and stop times, specified as "MustMatch". Specify StartStop="MustMatch" when you want the comparison to be sensitive to mismatches in signal start and stop times. When you specify this name-value argument, the algorithm compares the start and stop times for aligned signals before synchronizing and comparing the signal data.

When the start times and stop times do not match, the Status property of the Simulink.sdi.DiffSignalResult object for the result is set to StartStopMismatch.

When you specify that start and stop times must match and configure the comparison to stop on the first mismatch, a start or stop time mismatch stops the comparison. A stopped comparison may not compute results for all signals.

`StopOnFirstMismatch` — Whether comparison stops on first detected mismatch
`"Metadata"` | `"Any"`

Whether comparison stops on first detected mismatch without comparing remaining signals, specified as `"Metadata"` or `"Any"`. A stopped comparison may not compute results for all signals, and can return a mismatched result more quickly.

- `"Metadata"` — A mismatch in metadata for aligned signals stops the comparison. Metadata comparisons happen before comparing signal data.

  The Simulation Data Inspector always aligns signals and compares signal units. When you configure the comparison to stop on the first mismatch, an unaligned signal or mismatched units always stop the comparison. You can specify additional name-value arguments to configure the comparison to check and stop on the first mismatch for additional metadata, such as signal data type, start and stop times, and time vectors.
- `"Any"` — A mismatch in metadata or signal data for aligned signals stops the comparison.

`ExpandChannels` — Whether to compute comparison results for each channel in multidimensional signals
`true` or `1` (default) | `false` or `0`

Whether to compute comparison results for each channel in multidimensional signals, specified as logical `true` (`1`) or `false` (`0`).

- `true` or `1` — Comparison expands multidimensional signals represented as a single signal with nonscalar sample values to a set of signals with scalar sample values and computes a comparison result for each signal.

  The representation of the multidimensional signal in the Simulation Data Inspector as a single signal with nonscalar sample values does not change.
- `false` or `0` — Comparison does not compute results for multidimensional signals represented as a single signal with nonscalar sample values.

Output Arguments

`diffResult` — Comparison results
`Simulink.sdi.DiffRunResult` object

Comparison results, returned as a `Simulink.sdi.DiffRunResult` object.

Limitations

The Simulation Data Inspector does not support comparing:

- Signals of data types `int64` or `uint64`.
- Variable-size signals.

Version History

Introduced in R2011b

See Also

Functions
`Simulink.sdi.compareSignals` | `Simulink.sdi.getRunIDByIndex` | `Simulink.sdi.getRunCount` | `getResultByIndex`

Objects
`Simulink.sdi.DiffRunResult` | `Simulink.sdi.DiffSignalResult`

Topics
"Inspect and Compare Data Programmatically"
"Compare Simulation Data"
"How the Simulation Data Inspector Compares Data"

sin

Sine of fixed-point values

Syntax

`y = sin(theta)`

Description

`y = sin(theta)` returns the sine of `fi` input `theta` using a lookup table algorithm.

Examples

Calculate the Sine of Fixed-Point Input Values

```
theta = fi([-pi/2,-pi/3,-pi/4,0,pi/4,pi/3,pi/2]);
y = sin(theta)
```

```
y = 
   -1.0000   -0.8661   -0.7072         0    0.7070    0.8659    0.9999

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Input Arguments

`theta` — Input angle in radians
real-valued `fi` object

Input angle in radians, specified as a real-valued `fi` object. `theta` can be a signed or unsigned scalar, vector, matrix, or multidimensional array containing the fixed-point angle values in radians. Valid data types of `theta` are:

- `fi` single
- `fi` double
- `fi` fixed-point with binary point scaling
- `fi` scaled double with binary point scaling

Data Types: `fi`

Output Arguments

y — Sine of input angle
scalar | vector | matrix | multidimensional array

Sine of input angle, returned as a scalar, vector, matrix, or multidimensional array. `y` is a signed, fixed-point number in the range [-1,1].

If the `DataTypeMode` property of `theta` is `Fixed-point: binary point scaling`, then `y` is returned as a signed fixed-point data type with binary point scaling, a 16-bit word length, and a 15-bit fraction length (`numerictype(1,16,15)`). If `theta` is a `fi` single, `fi` double, or `fi` scaled double with binary point scaling, then `y` is returned with the same data type as `theta`.

More About

Sine

The sine of angle Θ is defined as

$$\sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i}$$

Algorithms

The `sin` function computes the sine of fixed-point input using an 8-bit lookup table as follows:

1. Perform a modulo 2π, so the input is in the range [0,2π) radians.
2. Cast the input to a 16-bit stored integer value, using the 16 most-significant bits.
3. Compute the table index, based on the 16-bit stored integer value, normalized to the full `uint16` range.
4. Use the 8 most-significant bits to obtain the first value from the table.
5. Use the next-greater table value as the second value.
6. Use the 8 least-significant bits to interpolate between the first and second values, using nearest-neighbor linear interpolation.

fimath Propagation Rules

The `sin` function ignores and discards any `fimath` attached to the input, `theta`. The output, `y`, is always associated with the default `fimath`.

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`sin` | `angle` | `cos` | `atan2` | `cordicsin` | `cordiccos`

sign

Perform sign function (signum function) on array

Syntax

```
c = sign(a)
```

Description

`c = sign(a)` returns an array `c` the same size as `a`, where each element of `c` is:

- `1` if the corresponding element of `a` is greater than `0`.
- `0` if the corresponding element of `a` is `0`.
- `-1` if the corresponding element of `a` is less than `0`.

The elements of `c` are of data type `int8`.

Examples

Find Sign Function

Find the sign function of a `fi` object.

```
sign(fi(2))
```

```
ans = 

  int8

   1
```

Find the sign function of a signed `fi` vector.

```
v = fi([-11 0 1.5],1);
sign(v)
```

```
ans = 

  1×3 int8 row vector

   -1    0    1
```

Find the sign function of an unsigned `fi` vector.

```
u = fi([-11 0 1.5],0);
sign(u)
```

```
ans = 

  1×3 int8 row vector
```

```
   0   0   1
```

Input Arguments

a — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a `fi` scalar, vector, matrix, or multidimensional array.

`sign` does not support complex `fi` inputs.

Data Types: `fi`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`abs` | `complex` | `conj`

single

Single-precision floating-point real-world value of `fi` object

Syntax

```
single(a)
```

Description

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

`single(a)` returns the real-world value of a `fi` object in single-precision floating point.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For the automated workflow, do not use explicit double or single casts in your MATLAB algorithm to insulate functions that do not support fixed-point data types. The automated conversion tool does not support these casts. Instead of using casts, supply a replacement function. For more information, see “Function Replacements”.

See Also

`double`

sort

Sort elements of real-valued `fi` object in ascending or descending order

Syntax

```
B = sort(A)
B = sort(A,dim)
B = sort( ___ ,direction)
[B,I] = sort( ___ )
```

Description

`B = sort(A)` sorts the elements of the real-valued `fi` object `A` in ascending order.

- If `A` is a vector, then `sort(A)` sorts the vector elements.
- If `A` is a matrix, then `sort(A)` treats the columns of `A` as vectors and sorts each column.
- If `A` is a multidimensional array, then `sort(A)` operates along the first array dimension whose size does not equal `1`, treating the elements as vectors.

`B = sort(A,dim)` returns the sorted elements of `A` along dimension `dim`.

`B = sort( ___ ,direction)` returns sorted elements of `A` in the order specified by `direction`.

`[B,I] = sort( ___ )` also returns a collection of index vectors for any of the previous syntaxes.

Examples

Sort `fi` Vector in Ascending Order

Create a `fi` row vector and sort its elements in ascending order.

```
A = fi([9 0 -7 5 3 8 -10 4 2]);
B = sort(A)
```

```
B = 

   -10    -7     0     2     3     4     5     8     9

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Sort `fi` Matrix Columns in Descending Order

Create a matrix of `fi` values and sort its columns in descending order.

```
A = fi([10 -12 4 8; 6 -9 8 0; 2 3 11 -2; 1 1 9 3]);
B = sort(A,'descend')

B =

    10     3    11     8
     6     1     9     3
     2    -9     8     0
     1   -12     4    -2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Sort and Index a fi Matrix

Create a matrix of `fi` values and sort each of its rows in ascending order.

```
A = fi([3 6 5; 7 -2 4; 1 0 -9]);
[B,I] = sort(A,2)

B =

     3     5     6
    -2     4     7
    -9     0     1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11

I =

  3×3 int32 matrix

   1   3   2
   2   3   1
   3   2   1
```

`B` contains the sorted values and `I` is a collection of 1-by-3 row index vectors describing the rearrangement of each row of `A`.

Input Arguments

A — Input array
real-valued `fi` object

Input array, specified as a real-valued `fi` object.

- If `A` is a scalar, then `sort(A)` returns `A`.
- If `A` is a vector, then `sort(A)` sorts the vector elements.
- If `A` is a matrix, then `sort(A)` treats the columns of `A` as vectors and sorts each column.

- If `A` is a multidimensional array, then `sort(A)` operates along the first array dimension whose size does not equal `1`, treating the elements as vectors.

`sort` does not support complex fixed-point inputs, or pairs of `Name,Value` arguments. Refer to the MATLAB `sort` reference page for more information.

Data Types: `fi`

`dim` — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal `1`.

The dimensions argument must be a built-in data type; it cannot be a `fi` object.

Example: Consider a matrix `A`. `sort(A,1)` sorts the elements in the columns of `A`.

Example: `sort(A,2)` sorts the elements in the rows of `A`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`direction` — Sorting direction
`'ascend'` (default) | `'descend'`

Sorting direction, specified as `'ascend'` or `'descend'`.

Data Types: `char`

Output Arguments

`B` — Sorted array
scalar | vector | matrix | multidimensional array

Sorted array, returned as a scalar, vector, matrix, or multidimensional array. `B` is the same size and type as `A`. The order of the elements in `B` preserves the order of any equal elements in `A`.

`I` — Sort index
scalar | vector | matrix | multidimensional array

Sort index, returned as a scalar, vector, matrix, or multidimensional array. `I` is the same size as `A`. The index vectors are oriented along the same dimension that `sort` operates on.

Example: If `A` is a vector, then `B = A(I)`.

Example: If `A` is a 2-by-3 matrix, then `[B,I] = sort(A,2)` sorts the elements in each row of `A`. The output `I` is a collection of 1-by-3 row index vectors describing the rearrangement of each row of `A`.

Version History

Introduced in R2008b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The dimensions argument must be a built-in type; it cannot be a `fi` object.

See Also

`sort`

Topics

"Reshaping and Rearranging Arrays"

sqrt

Square root of fi object

Syntax

```
c = sqrt(a)
c = sqrt(a,T)
c = sqrt(a,F)
c = sqrt(a,T,F)
```

Description

This function computes the square root of a fi object using a bisection algorithm.

c = sqrt(a) returns the square root of fi object a. Intermediate quantities are calculated using the fimath associated with a. The numerictype object of c is determined automatically using an “Internal Rule” on page 4-1014.

c = sqrt(a,T) returns the square root of fi object a with numerictype object T. Intermediate quantities are calculated using the fimath associated with a. See “Data Type Propagation Rules” on page 4-1014.

c = sqrt(a,F) returns the square root of fi object a. Intermediate quantities are calculated using the fimath object F. The numerictype object of c is determined automatically using an “Internal Rule” on page 4-1014.

When a is a built-in double or single data type, this syntax is equivalent to c = sqrt(a) and the fimath object F is ignored.

c = sqrt(a,T,F) returns the square root fi object a with numerictype object T. Intermediate quantities are also calculated using the fimath object F. See “Data Type Propagation Rules” on page 4-1014.

Input Arguments

a — Input fi array
scalar | vector | matrix | multidimensional array

Input fi array, specified as a scalar, vector, matrix, or multidimensional array.

sqrt does not support complex, negative-valued, or [Slope Bias] inputs.

Example: a = fi(pi,1,8,3)

Data Types: fi

T — numerictype of output
numerictype object

numerictype of the output c, specified as a numerictype object.

Example: T = numerictype(1,32,30)

F — `fimath` used for calculations of intermediate quantities
`fimath` object

`fimath` used for calculations of intermediate quantities, specified as a `fimath` object.

Example: `F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent')`

Algorithms

Internal Rule

For syntaxes where the `numerictype` object of the output is not specified as an input to the `sqrt` function, it is automatically calculated according to the following internal rule:

$$sign_c = sign_a$$

$$WL_c = \mathrm{ceil}(\frac{WL_a}{2})$$

$$FL_c = WL_c - \mathrm{ceil}(\frac{WL_a - FL_a}{2})$$

Data Type Propagation Rules

For syntaxes for which you specify a `numerictype` object `T`, the `sqrt` function follows the data type propagation rules listed in the following table. In general, these rules can be summarized as "floating-point data types are propagated." This allows you to write code that can be used with both fixed-point and floating-point inputs.

Data Type of Input fi Object a	Data Type of numerictype object T	Data Type of Output c
Built-in `double`	Any	Built-in `double`
Built-in `single`	Any	Built-in `single`
`fi Fixed`	`fi Fixed`	Data type of `numerictype` object `T`
`fi ScaledDouble`	`fi Fixed`	`ScaledDouble` with properties of `numerictype` object `T`
`fi double`	`fi Fixed`	`fi double`
`fi single`	`fi Fixed`	`fi single`
Any `fi` data type	`fi double`	`fi double`
Any `fi` data type	`fi single`	`fi single`

Version History

Introduced in R2006b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Complex and [Slope Bias] inputs error out.
- Negative inputs yield a 0 result for generated C code.
- Negative inputs error out for MATLAB Executable (MEX) code.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`fi` | `fimath` | `numerictype`

step

Process data and visualize dynamic range

Syntax

```
step(H, data)
```

Description

`step(H, data)` processes your data and allows you to visualize the dynamic range in the scope window. The `NumericTypeScope` retains previously collected information about the variable between each call to `step`. In order to process new data, call `reset` before calling `step` again.

Examples

View the Dynamic Range of a `fi` Object

Use the `NumericTypeScope` to view the dynamic range of a `fi` object.

Create a `fi` object and set the `DataTypeOverride` to `ScaledDoubles`.

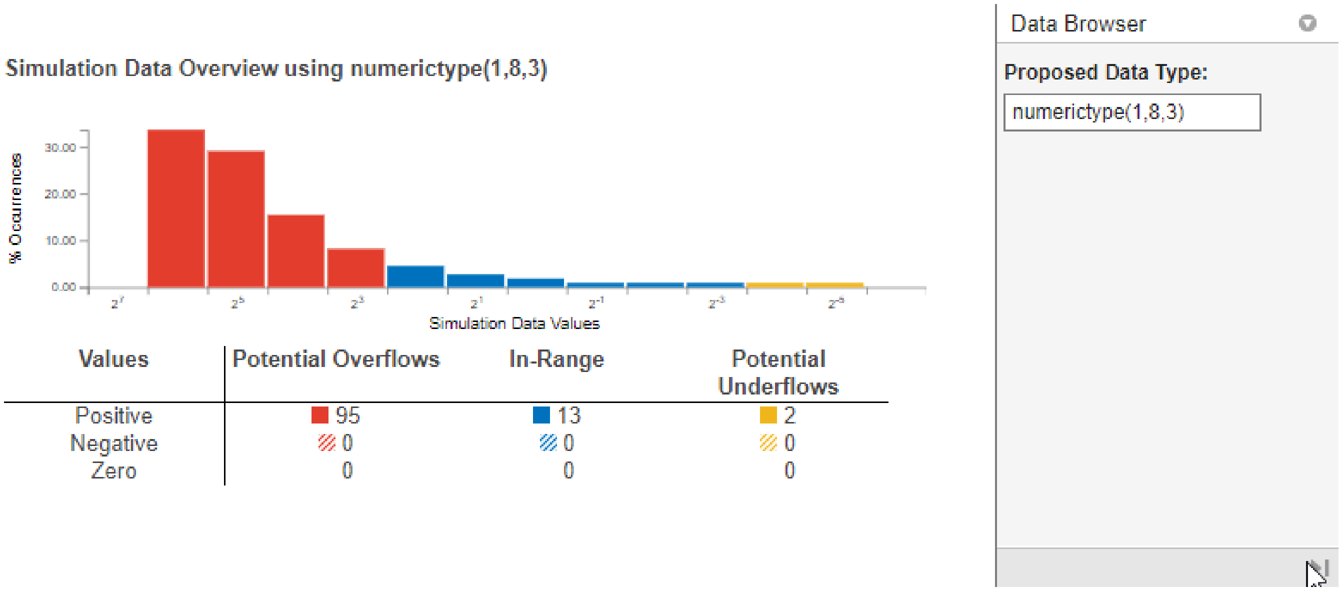
```
a = fi(magic(10),1,8,2);
b = fi([a; 2.^(-5:4)],1,8,3);

fp = fipref;
initialDTOSetting = fp.DataTypeOverride;
fp.DataTypeOverride = 'ScaledDoubles';
```

Create a `NumericTypeScope` object. You can use the `reset` method to ensure that all stored information is cleared from the `NumericTypeScope` object `h`.

```
h = NumericTypeScope;
reset(h)
```

Use the `step` method to process your data and visualize the dynamic range of the `fi` object `b`.

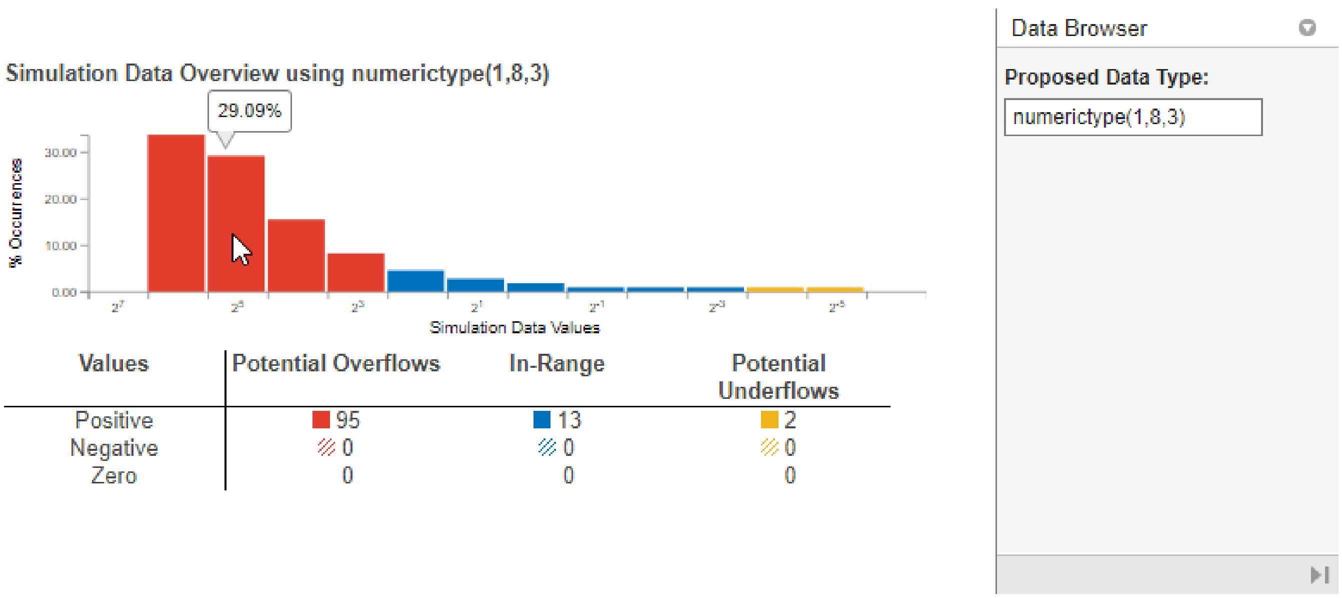
```
step(h,b);
```



Closing the NumericTypeScope window does not delete the object from your workspace. Close the NumericTypeScope window and reopen it using the show function.

```
show(h);
```

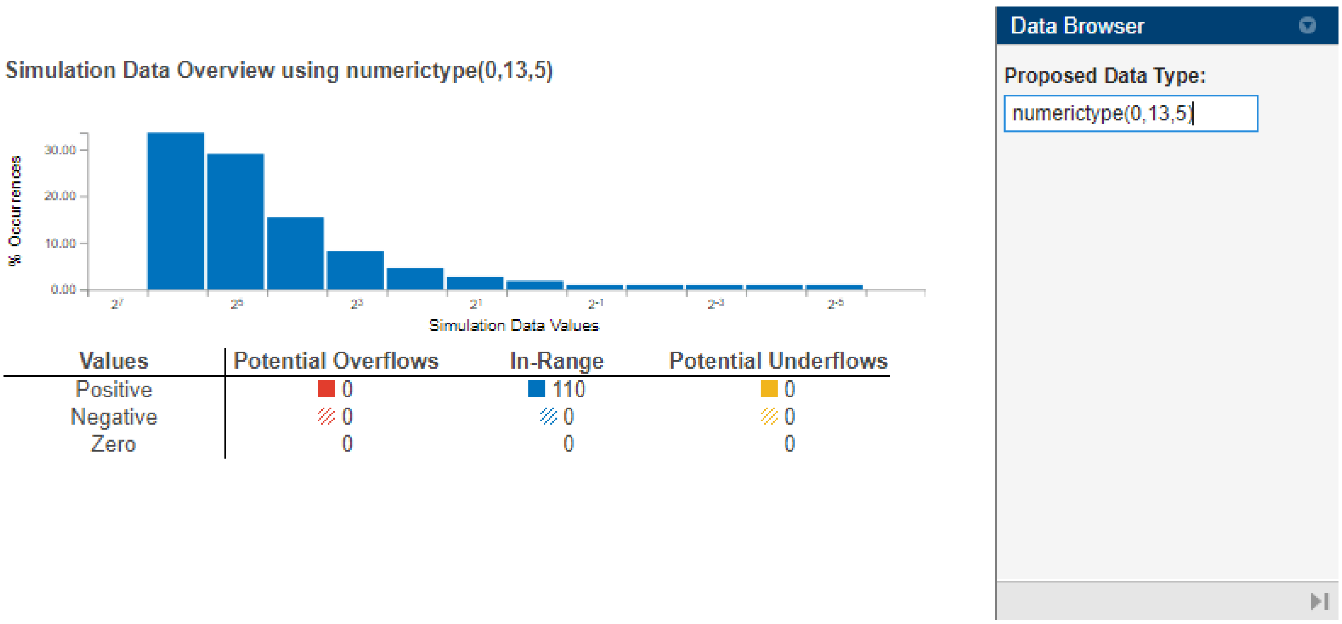
The NumericTypeScope displays a log2 histogram which shows that the values appear both outside of the range and below the precision of the data type of the variable. Pause on one bar of the histogram to view the percentage of the total values that are represented by that bar.



In this case, the data type of b is numerictype(1,8,3). The numerictype(1,8,3) data type provides 5 integer bits, including the signed bit, and 3 fractional bits. Thus, this data type can represent only values between -2^4 and 2^4 - 2^-3 (from -16 to 15.8750). Given the range and precision of this data type, values greater than 2^4 fall outside the range and values less than 2^-3 fall below the precision of the data type.

The NumericTypeScope shows that values requiring bits 5, 6, and 7 are outside the range and values requiring fractional bits 4 and 5 are below precision. Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable b to numerictype(0,13,5).

Given this information, you can prevent values that are outside range and below precision by changing the data type of the variable b to numerictype(0,13,5). In the NumericTypeScope, enter numerictype(0,13,5) in the **Proposed Data Type** box.



Return to the original data type override setting.

```
fp.DataTypeOverride = initialDTOSetting;
```

Input Arguments

H — NumericTypeScope object

NumericTypeScope object

NumericTypeScope object, specified as a NumericTypeScope object.

Example: step(H,a)

data — Data to process

fi object

Data to process, specified as a fi object.

Example: step(H,a)

Data Types: fi

Version History

Introduced in R2010a

See Also

`NumericTypeScope` | `show` | `reset`

storedInteger

Package: `embedded`

Stored integer value of `fi` object

Syntax

`x = storedInteger(a)`

Description

`x = storedInteger(a)` returns the stored integer value of `fi` object `a`.

Fixed-point numbers can be represented as

$$real\text{-}worldvalue = 2^{-fractionlength} \times storedinteger$$

or, equivalently as

$$real\text{-}worldvalue = (slope \times storedinteger) + bias$$

The stored integer is the raw binary number, in which the binary point is assumed to be at the far right of the word.

Examples

Stored Integer Value of `fi` Objects

This example shows how to find the stored integer values for two `fi` objects. Use the `class` function to display the stored integer data types.

```
x = fi([0.2 0.3 0.5 0.3 0.2]);
in_x = storedInteger(x);
c1 = class(in_x)
```

```
c1 =
'int16'
```

```
numtp = numerictype('WordLength',17);
x_n = fi([0.2 0.3 0.5 0.3 0.2],'numerictype',numtp);
in_xn = storedInteger(x_n);
c2 = class(in_xn)
```

```
c2 =
'int32'
```

Input Arguments

a — Fixed-point numeric object
`fi` object

Fixed-point numeric object from which you want to get the stored integer value, specified as a `fi` object.

Data Types: `fi`
Complex Number Support: Yes

Output Arguments

`x` — Stored integer value of `fi` object
integer

Stored integer value of `fi` object, returned as an integer.

The returned stored integer value is the smallest built-in integer data type in which the stored integer value `f` fits. Signed `fi` values return stored integers of type `int8`, `int16`, `int32`, or `int64`. Unsigned `fi` values return stored integers of type `uint8`, `uint16`, `uint32`, or `uint64`. The return type is determined based on the stored integer word length (WL):

- WL ≤ 8 bits, the return type is `int8` or `uint8`.
- 8 bits < WL ≤ 16 bits, the return type is `int16` or `uint16`.
- 16 bits < WL ≤ 32 bits, the return type is `int32` or `uint32`.
- 32 bits < WL ≤ 64 bits, the return type is `int64` or `uint64`.

Tips

When the word length is greater than 64 bits, the `storedInteger` function errors. For bit-true integer representation of very large word lengths, use `bin`, `oct`, `dec`, `hex`, or `sdec`.

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `storedIntegerToDouble`

storedIntegerToDouble

Package: `embedded`

Convert stored integer value of `fi` object to built-in double value

Syntax

`d = storedIntegerToDouble(a)`

Description

`d = storedIntegerToDouble(a)` converts the stored integer value of `fi` object, `a`, to a double-precision floating-point value, `d`.

If the input word length is greater than 52 bits, a quantization error may occur. `Inf` is returned if the stored integer value of the input `fi` object is outside the representable range of built-in double values.

Examples

Convert Stored Integer Value of `fi` Object to Double-Precision Value

Convert the stored integer of a `fi` value to a double-precision value. Use the `class` function to verify that the stored integer is a double-precision value.

```
a = fi(pi,1,16,12)
```

```
a = 
    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

```
d = storedIntegerToDouble(a)
```

```
d = 12868
```

```
dtype = class(d)
```

```
dtype = 
'double'
```

Input Arguments

a — Value to convert
`fi` object

Value to convert, specified as a `fi` object.

Data Types: fi
Complex Number Support: Yes

Version History

Introduced in R2012a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

storedInteger | fi | class

stripscaling

Stored integer of `fi` object

Syntax

`I = stripscaling(a)`

Description

`I = stripscaling(a)` returns the stored integer of `a` as a `fi` object with binary-point scaling, zero fraction length and the same word length and sign as `a`.

Examples

`stripscaling` is useful for converting the value of a `fi` object to its stored integer value.

```
fipref('NumericTypeDisplay','short', ...
       'FimathDisplay','none');
format long g
a = fi(0.1,true,48,47)
```

```
a = 

         0.100000000000001
      numerictype(1,48,47)
```

```
b = stripscaling(a)
```

```
b = 

            140737488355333
      numerictype(1,48,0)
```

```
bin(a)
```

```
ans =

    '000011001100110011001100110011001100110011001101'
```

```
bin(b)
```

```
ans =

    '000011001100110011001100110011001100110011001101'
```

Notice that the stored integer values of `a` and `b` are identical, while their real-world values are different.

Version History

Introduced before R2006a

svd

Package: `embedded`

Fixed-point Golub-Kahan-Reinsch singular value decomposition

Syntax

```
S = svd(A)
[U,S,V] = svd(A)
[ ___ ] = svd(A,"econ")
[ ___ ] = svd(A,0)
[ ___ ] = svd( ___ ,sigmaForm)
```

Description

`S = svd(A)` returns the singular values of matrix `A` in descending order.

`[U,S,V] = svd(A)` performs a singular value decomposition of matrix `A`, such that `A = U*S*V'`. `S` is a diagonal matrix of the same dimension as `A` with nonnegative diagonal elements in decreasing order. `U` and `V` are unitary matrices.

`[ ___ ] = svd(A,"econ")` produces an economy-size decomposition of `A`. If `A` is an *m*-by-*n* matrix, then:

- *m* > *n* — Only the first *n* columns of `U` are computed and `S` is *n*-by-*n*.
- *m* = *n* — `svd(A,"econ")` is equivalent to `svd(A)`.
- *m* < *n* — Only the first *m* columns of `V` are computed, and `S` is *m*-by-*m*.

`[ ___ ] = svd(A,0)` produces a different economy-size decomposition of `A`. If `A` is an *m*-by-*n* matrix, then:

- *m* > *n* — `svd(A,0)` is equivalent to `svd(A,"econ")`.
- *m* <= *n* — `svd(A,0)` is equivalent to `svd(A)`.

Syntax is not recommended. Use the `"econ"` option instead.

`[ ___ ] = svd( ___ ,sigmaForm)` optionally specifies the output format for the singular values. You can use this option with any of the previous input or output combinations. Specify `"vector"` to return the singular values as a column vector. Specify `"matrix"` to return the singular values in a diagonal matrix.

Examples

Singular Values of Fixed-Point Matrix

Compute the singular values of a full rank fixed-point matrix.

```
A = fi([1 0 1; -1 -2 0; 0 1 -1])
```

```
A = 
     1     0     1
    -1    -2     0
     0     1    -1

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14
```

Compute the singular values.

```
s = svd(A)
```

```
s = 
    2.4605
    1.6996
    0.2391

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16
```

The singular values are returned in a column vector in decreasing order.

Fixed-Point Singular Value Decomposition

Find the singular value decomposition of the rectangular fixed-point matrix A.

Define the rectangular matrix A.

```
m = 4;
n = 2;
rng('default');
A = fi(10*randn(m,n))
```

```
A = 
    5.3770    3.1875
   18.3389  -13.0771
  -22.5889   -4.3359
    8.6221    3.4258

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10
```

Find the singular value decomposition of the fixed-point matrix A.

```
[U,S,V] = svd(A)
```

```
U = 
    0.1590    0.2717   -0.9387   -0.1403
    0.6397   -0.7548   -0.1219    0.0790
   -0.7049   -0.5057   -0.3224    0.3786
```

```
    0.2619    0.3174         0    0.9114

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S =
   31.0148         0
         0   14.1292
         0         0
         0         0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16

V =
    0.9920    0.1259
   -0.1259    0.9920

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Confirm the relation `A = U*S*V'`.

```
U*S*V'
```

```
ans =
    5.3770    3.1875
   18.3390  -13.0774
  -22.5890   -4.3361
    8.6221    3.4258

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 99
        FractionLength: 76
```

Economy-Size Decomposition

Calculate the complete and economy-size decomposition of a rectangular fixed-point matrix.

Define the fixed-point matrix `A`.

```
m = 5;
n = 3;
rng('default');
A = fi(10*randn(m,n))
```

```
A =
    5.3770  -13.0762  -13.4980
   18.3379   -4.3359   30.3496
```

```
  -22.5879    3.4258    7.2539
    8.6211   35.7832   -0.6309
    3.1875   27.6953    7.1465

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 9
```

Compute the complete decomposition.

```
[U,S,V] = svd(A)
```

```
U =
    0.3081   -0.0950    0.4507    0.7929    0.2534
   -0.1437    0.9533   -0.0877    0.2415   -0.0675
   -0.0224   -0.2106   -0.8423    0.4887   -0.0831
   -0.7299   -0.1909    0.2773    0.2722   -0.5290
   -0.5926   -0.0375   -0.0541         0    0.8028

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S =
   48.4486         0         0
         0   36.6721         0
         0         0   26.9112
         0         0         0
         0         0         0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16

V =
   -0.1786    0.5443    0.8196
   -0.9497   -0.3131    0.0010
   -0.2571    0.7783   -0.5729

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Compute the economy-size decomposition.

```
[U,S,V] = svd(A,"econ")
```

```
U =
    0.3082   -0.0950    0.4507
   -0.1437    0.9533   -0.0878
   -0.0224   -0.2106   -0.8423
   -0.7299   -0.1909    0.2773
   -0.5926   -0.0375   -0.0541

          DataTypeMode: Fixed-point: binary point scaling
```

```
           Signedness: Signed
           WordLength: 32
       FractionLength: 30

S = 
   48.4487         0         0
         0   36.6720         0
         0         0   26.9111

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16

V = 
   -0.1786    0.5444    0.8196
   -0.9497   -0.3131    0.0010
   -0.2571    0.7782   -0.5729

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Use the expected result `A = U*S*V'` to determine the relative error of the calculation.

```
relativeError = norm(double(U*S*V'-A))/norm(double(A))
```

```
relativeError = 1.3361e-05
```

Control Singular Value Output Format

Create a 3-by-3 magic square matrix and calculate the singular value decomposition. By default, the `svd` function returns the singular values in a diagonal matrix when you specify multiple outputs.

Define the matrix `A`.

```
m = 3;
n = m;
A = fi(magic(m))
```

```
A = 
     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Compute the singular value decomposition.

```
[U,S,V] = svd(A)
```

```
U = 
    0.5774   -0.7071   -0.4082
```

```
    0.5773    0.0000    0.8165
    0.5773    0.7071   -0.4083

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S =
   15.0000         0         0
         0    6.9282         0
         0         0    3.4640

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16

V =
    0.5774   -0.4083   -0.7071
    0.5773    0.8165   -0.0000
    0.5773   -0.4082    0.7071

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30
```

Specify the `"vector"` option to return the singular values in a column vector.

```
[U,S,V] = svd(A,"vector")
```

```
U =
    0.5774   -0.7071   -0.4082
    0.5773    0.0000    0.8165
    0.5773    0.7071   -0.4083

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 30

S =
   15.0000
    6.9282
    3.4640

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16

V =
    0.5774   -0.4083   -0.7071
    0.5773    0.8165   -0.0000
    0.5773   -0.4082    0.7071

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
        WordLength: 32
    FractionLength: 30
```

If you specify one output argument, such as `S = svd(A)`, then `svd` switches behavior to return the singular values in a column vector by default. In that case, you can specify the `"matrix"` option to return the singular values as a diagonal matrix.

Input Arguments

A — Input matrix
matrix

Input matrix, specified as a matrix. `A` can be a fixed-point or scaled double `fi` data type.

Data Types: `fi`
Complex Number Support: Yes

sigmaForm — Output format of singular values
`"vector"` | `"matrix"`

Output format of singular values, specified as one of these values:

- `"vector"` — `S` is a column vector. This behavior is the default when you specify one output, `S = svd(A)`.
- `"matrix"` — `S` is a diagonal matrix. This behavior is the default when you specify multiple outputs, `[U,S,V] = svd(A)`.

Example: `[U,S,V] = svd(X,"vector")` returns `S` as a column vector instead of a diagonal matrix.

Example: `S = svd(X,"matrix")` returns `S` as a diagonal matrix instead of a column vector.

Data Types: `char` | `string`

Output Arguments

U — Left singular vectors
matrix

Left singular vectors, returned as the columns of a matrix.

The fixed-point data type is adjusted to avoid overflow and increase precision. For more information, see "Algorithms" on page 4-1032.

S — Singular values
diagonal matrix | column vector

Singular values, returned as a diagonal matrix or column vector. The singular values are nonnegative and returned in decreasing order.

The fixed-point data type is adjusted to avoid overflow and increase precision. For more information, see "Algorithms" on page 4-1032.

V — Right singular vectors
matrix

Right singular vectors, returned as the columns of a matrix.

The fixed-point data type is adjusted to avoid overflow and increase precision. For more information, see “Algorithms” on page 4-1032.

Tips

To have full control over the fixed-point types, use the `fixed.svd` function.

Algorithms

Data Type Propagation

The `svd` function adjusts the data type of a fixed-point input to avoid overflow and increase precision. The fraction length of the singular vectors `S` is adjusted to a minimum of `16`, and the word length is increased to avoid overflow with a minimum of `32`. The word length of the left and right singular vectors `U` and `V` are the same as the word length of `S`. The fraction length of `U` and `V` is two less than the word length.

Golub-Kahan-Reinsch

The Golub-Kahan-Reinsch algorithm is a sequential method that performs well on serial computers. For parallel computing, as in FPGA and ASIC applications, use the `fixed.jacobiSVD` function.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

`svd` generates efficient, purely integer C code.

See Also

`fixed.svd` | `svd`

Topics

“Singular Values”

sub

Subtract two `fi` objects using `fimath` object

Syntax

```
c = sub(F,a,b)
```

Description

`c = sub(F,a,b)` subtracts `fi` objects `a` and `b` using `fimath` object `F`. This is helpful in cases when you want to override the `fimath` objects of `a` and `b`, or if the `fimath` properties associated with `a` and `b` are different. The output of `fi` object `c` has no local `fimath`.

Examples

Subtract Two `fi` Objects Overriding Their `fimath`

```
a = fi(pi);
b = fi(exp(1));
F = fimath('SumMode','SpecifyPrecision',...
    'SumWordLength',32,'SumFractionLength',16);
c = sub(F,a,b)
```

```
c =
    0.4233

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 16
```

`c` is the 32-bit difference of `a` and `b`, with fraction length 16.

Input Arguments

F — `fimath`
`fimath` object

`fimath` object to use for subtraction, specified as a `fimath` object.

`a,b` — Operands
scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays.

`a` and `b` must both be `fi` objects and must have the same dimensions unless one is a scalar. If either `a` or `b` is scalar, then `c` has the dimensions of the nonscalar object.

Data Types: `fi`

Complex Number Support: Yes

Algorithms

```
C = sub(F,A,B)
```

or

```
C = F.sub(A,B)
```

is equivalent to

```
A.fimath = F;
B.fimath = F;
C = A - B;
```

except that the `fimath` properties of `A` and `B` are not modified when you use the functional form.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation does not support the syntax `F.sub(a,b)`. You must use the syntax `sub(F,a,b)`.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`add` | `divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide`

subsasgn

Package: `embedded`

Subscripted assignment

Syntax

```
A = subsasgn(A,S,B)
```

Description

`A = subsasgn(A,S,B)` is called for the syntax `A(i) = B`, `A{i} = B`, or `A.i = B` when `A` is an object.

MATLAB uses the built-in `subsasgn` function to interpret indexed assignment statements:

- `A(i) = B` assigns the values of `B` into the elements of `A` specified by the subscript vector `i`. `B` must have the same number of elements as `i` or be a scalar value.
- `A(i,j) = B` assigns the values of `B` into the elements of the rectangular submatrix of `A` specified by the subscript vectors `i` and `j`. `B` must have `length(i)` rows and `length(j)` columns.
- A colon used as a subscript, as in `A(i,:) = B` or `A(:,i) = B`, indicates the entire column or row.
- For multidimensional arrays, `A(i,j,k,...) = B` assigns `B` to the specified elements of `A`. `B` must be `length(i)`-by-`length(j)`-by-`length(k)`-... or be shiftable to that size by adding or removing singleton dimensions.

Tip You can use fixed-point assignment, for example, `A(:) = B`, to cast a value with one numeric type into another numeric type. This subscripted assignment statement assigns the value of `B` into `A` while keeping the numeric type of `A`. Subscripted assignment works the same way for integer data types.

Note You must call `subsasgn` with an output argument. `subsasgn` does not modify the object used in the indexing operation (the first argument). You must assign the output to obtain a modified object.

Examples

Cast 16-bit Number into 8-bit Number

For `fi` objects `a` and `b`, there is a difference between

```
a = b
```

and

`a(:) = b`.

In the first case, `a = b` replaces `a` with `b` while `a` assumes the value, numeric type, and `fimath` object associated with `b`. In the second case, `a(:) = b` assigns the value of `b` into `a` while keeping the numeric type of `a`. You can use this to cast a value with one `numerictype` object into another `numerictype` object.

For example, cast a 16-bit number into an 8-bit number.

```
a = fi(0, 1, 8, 7)

a =
     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7

b = fi(pi/4, 1, 16, 15)

b =
    0.7854

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15

a(:) = b

a =
    0.7891

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 7
```

Emulate 40-bit Accumulator of a DSP

Define the variable `acc` to emulate a 40-bit accumulator of a DSP. The products and sums in this example are assigned into the accumulator using the syntax `acc(1)=...`. Assigning values into the accumulator is like storing a value in a register. To begin, turn on the logging mode and define the variables. In this example, `n` is the number of points in the input data `x` and output data `y`, and `t` represents time. The remaining variables are all defined as `fi` objects. The input data `x` is a high-frequency sinusoid added to a low-frequency sinusoid.

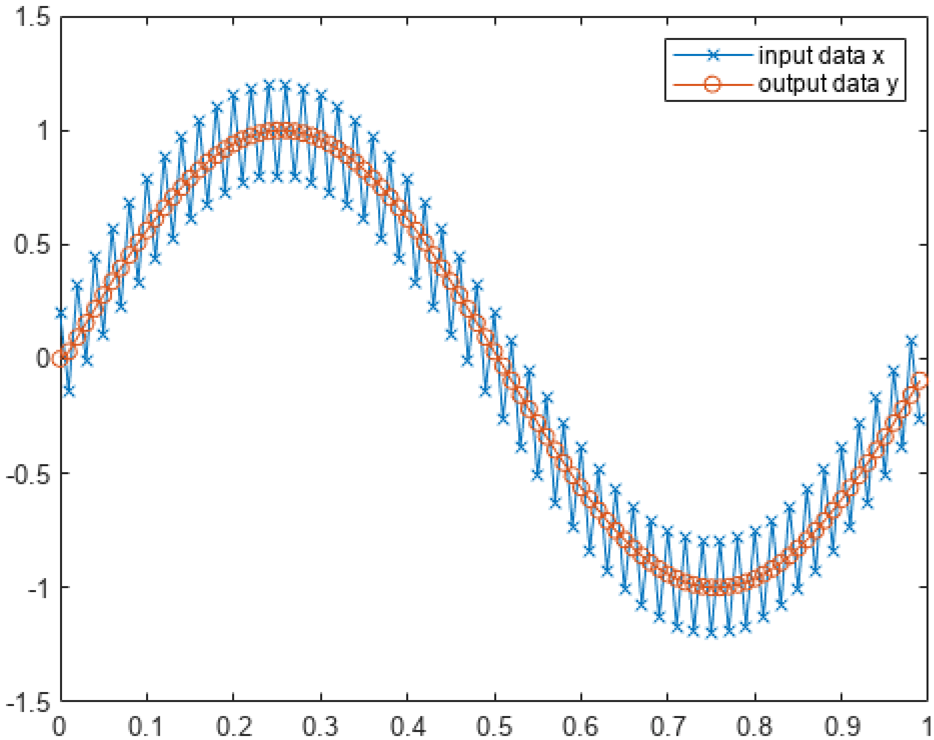
```
fipref('LoggingMode', 'on');
n = 100;
t = (0:n-1)/n;
x = fi(sin(2*pi*t) + 0.2*cos(2*pi*50*t));
b = fi([.5 .5]);
y = zeros(size(x),'like',x);
acc = fi(0.0, true, 40, 30);
```

The following loop takes a running average of the input `x` using the coefficients in `b` . Notice that `acc` is assigned into `acc(1)=...` versus using `acc=...`, which would overwrite and change the data type of `acc` .

```
for k = 2:n
    acc(1) = b(1)*x(k);
    acc(1) = acc + b(2)*x(k-1);
    y(k) = acc;
end
```

By averaging every other sample, the loop shown above passes the low-frequency sinusoid through and attenuates the high-frequency sinusoid.

```
plot(t,x,'x-',t,y,'o-')
legend('input data x','output data y')
```



The log report shows the minimum and maximum logged values and ranges of the variables used. Because `acc` is assigned into rather than overwritten, these logs reflect the accumulated minimum and maximum values.

```
logreport(x, y, b, acc)
```

```
              minlog        maxlog    lowerbound    upperbound    noverflows    nunde
      x    -1.200012      1.197998            -2      1.999939             0
      y   -0.9990234     0.9990234            -2      1.999939             0
      b          0.5           0.5            -1     0.9999695             0
    acc   -0.9990234     0.9989929          -512           512             0
```

Display `acc` to verify that its data type did not change.

```
acc
```

```
acc = 
   -0.0941

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 40
        FractionLength: 30
```

Reset the `fipref` object to restore its default values.

```
reset(fipref)
```

Input Arguments

A — Object used in indexing operation
scalar | vector | multidimensional array

Object used in indexing operation, specified as a scalar, vector, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

S — Type of indexing and subscripts
structure array

Type of indexing and subscripts, specified as a structure array. `S` is a structure array with two fields:

- `type` is a character vector or string containing `()`, `{}`, or `.`, specifying the subscript type.
- `subs` is a cell array, character array, or string array containing the actual subscripts.

Example: The syntax `A(1:2,:) = B` calls `a = subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type = '()'` and `S.subs = {1:2, ':'}`. A colon used as a script is passed as `':'`.

Data Types: `struct`

B — Value being assigned
scalar | vector | multidimensional array

Value being assigned, specified as a scalar, vector, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`
Complex Number Support: Yes

Output Arguments

A — Result of assignment statement
scalar | vector | multidimensional array

Result of assignment statement, which is the modified object passed in as the first argument, returned as a scalar, vector, or multidimensional array.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” (HDL Coder).

See Also

`subsref` | `cast`

Topics
“Cast fi Objects”
“Manual Fixed-Point Conversion Best Practices”

subsref

Subscripted reference

Description

This function accepts `fi` objects as inputs.

Refer to the MATLAB `subsref` reference page for more information.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

Supported data types for HDL code generation are listed in “Supported MATLAB Data Types, Operators, and Control Flow Statements” (HDL Coder).

sum

Sum of fi array elements

Syntax

```
S = sum(A)
S = sum(A,dim)
S = sum( ___ ,type)
```

Description

`S = sum(A)` returns the sum along different dimensions of the `fi` array `A`.

- If `A` is a vector, `sum(A)` returns the sum of the elements.
- If `A` is a matrix, `sum(A)` treats the columns of `A` as vectors, returning a row vector of the sums of each column.
- If `A` is a multidimensional array, `sum(A)` treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.

`S = sum(A,dim)` sums along the dimension `dim` of `A`.

`S = sum( ___ ,type)` returns an array in the class specified by `type`.

Examples

Sum of Vector Elements

Create a `fi` vector and specify `fimath` properties in the constructor.

```
A =  fi([1 2 5 8 5], 'SumMode', 'KeepLSB', 'SumWordLength', 32)
```

```
A = 
     1     2     5     8     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: KeepLSB
         SumWordLength: 32
         CastBeforeSum: true
```

Compute the sum of the elements of `A`.

```
S = sum(A)
```

```
S = 
    21

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 11

        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: KeepLSB
         SumWordLength: 32
         CastBeforeSum: true
```

The output `S` is a scalar with the specified `SumWordLength` of `32`. The `FractionLength` of `S` is `11` because `SumMode` was set to `KeepLSB`.

Sum of Elements in Each Column

Create a `fi` array, and compute the sum of the elements in each column.

```
A=fi([1 2 8;3 7 0;1 2 2])
```

```
A = 
     1     2     8
     3     7     0
     1     2     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
S=sum(A)
```

```
S = 
     5    11    10

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 11
```

MATLAB® returns a row vector with the sums of each column of `A`. The `WordLength` of `S` has increased by two bits because `ceil(log2(size(A,1)))=2`. The `FractionLength` remains the same because the default setting of `SumMode` is `FullPrecision`.

Sum of Elements in Each Row

Compute the sum along the second dimension (`dim=2`) of 3-by-3 matrix `A`.

```
A=fi([1 2 8;3 7 0;1 2 2])
```

```
A = 
     1     2     8
     3     7     0
     1     2     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
S=sum(A, 2)
```

```
S = 
    11
    10
     5

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 11
```

MATLAB® returns a column vector of the sums of the elements in each row. The `WordLength` of `S` is 18 because `ceil(log2(size(A,2)))=2`.

Sum of Elements Preserving Data Type

Compute the sums of the columns of `A` so that the output array, `S`, has the same data type.

```
A = fi([1 2 8;3 7 0;1 2 2])
```

```
A = 
     1     2     8
     3     7     0
     1     2     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

```
class(A)
```

```
ans = 
'embedded.fi'
```

```
S = sum(A, 'native')
```

```
S = 
     5    11    10

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 11
```

```
class(S)
```

```
ans =
'embedded.fi'
```

MATLAB® preserves the data type of A and returns a row vector S of type embedded.fi.

Input Arguments

A — Input fi array
fi object | numeric variable

fi input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

dim — Dimension to operate along
positive integer scalar

Dimension to operate along, specified as a positive integer scalar. dim can also be a fi object. If no value is specified, the default is the first array dimension whose size does not equal 1.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

type — Output class
'double' | 'native'

Output class, specified as 'double' or 'native'. The output class defines the data type that the operation is performed in and returned in.

- If type is 'double', then sum returns a double-precision array, regardless of the input data type.
- If type is 'native', then sum returns an array with the same class as input array A.

Data Types: char

Output Arguments

S — Sum array
scalar | vector | matrix | multidimensional array

Sum array, returned as a scalar, vector, matrix, or multidimensional array.

Note The fimath object is used in the calculation of the sum. If SumMode is set to FullPrecision, KeepLSB, or KeepMSB, then the number of integer bits of growth for sum(A) is ceil(log2(size(A,dim))).

Limitations

- sum does not support fi objects of data type Boolean.

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Variable-sized inputs are only supported when the `SumMode` property of the governing `fimath` object is set to `SpecifyPrecision` or `KeepLSB`.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`sum` | `add` | `divide` | `fi` | `fimath` | `mpy` | `mrdivide` | `numerictype` | `rdivide` | `sub`

times, .*

Package: embedded

Element-by-element multiplication of fi objects

Syntax

```
C = A.*B
C = times(A,B)
```

Description

C = A.*B performs element-by-element multiplication of A and B, and returns the result in C.

times does not support fi objects of data type boolean.

C = times(A,B) is an alternate way to execute A.*B.

Examples

Multiply a fi Object by a Scalar

Use the times function to perform element-by-element multiplication of a fi object and a scalar.

```
a=4;
b=fi([2 4 7; 9 0 2])

b = 
     2     4     7
     9     0     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

a is a scalar double, and b is a matrix of fi objects. When doing arithmetic between a fi and a double, the double is cast to a fi with the same word length and signedness of the fi, and best-precision fraction length. The result of the operation is a fi.

```
c=a.*b

c = 
     8    16    28
    36     0     8

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 23
```

During the operation, a was cast to a fi object with wordlength 16. The output, c, is a fi object with word length 32, the sum of the word lengths of the two multiplicands, a and b. This is because the default setting of ProductMode in fimath is FullPrecision.

Multiply Two fi Objects

Use the times function to perform element-by-element multiplication of two fi objects.

```
a=fi([5 9 9; 1 2 -3], 1, 16, 3)
```

```
a = 
     5     9     9
     1     2    -3

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 3
```

```
b=fi([2 4 7; 9 0 2], 1, 16, 3)
```

```
b = 
     2     4     7
     9     0     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 3
```

```
c=a.*b
```

```
c = 
    10    36    63
     9     0    -6

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 32
        FractionLength: 6
```

The word length and fraction length of c are equal to the sums of the word lengths and fraction lengths of a and b. This is because the default setting of ProductMode in fimath is FullPrecision.

Input Arguments

A — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of fi objects or built-in data types. Inputs A and B must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

`times` does not support `fi` objects of data type `boolean`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

B — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array of `fi` objects or built-in data types. Inputs `A` and `B` must either be the same size or have sizes that are compatible. For more information, see “Compatible Array Sizes for Basic Operations”.

`times` does not support `fi` objects of data type `boolean`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

R2021b: Implicit expansion change affects arguments for operators

Behavior changed in R2021b

Starting in R2021b with the addition of implicit expansion for `fi` `times`, `plus`, and `minus`, some combinations of arguments for basic operations that previously returned errors now produce results.

If your code uses element-wise operators and relies on the errors that MATLAB previously returned for mismatched sizes, particularly within a `try/catch` block, then your code might no longer catch those errors.

For more information on the required input sizes for basic array operations, see “Compatible Array Sizes for Basic Operations”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Any non-`fi` input must be constant; that is, its value must be known at compile time so that it can be cast to a `fi` object.
- When you provide complex inputs to the `times` function inside of a MATLAB Function block, you must declare the input as complex before running the simulation. To do so, go to the Model Explorer and set the **Complexity** parameter for all known complex inputs to `On`.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

plus | minus | mtimes | uminus

toeplitz

Create Toeplitz matrix

Syntax

```
t = toeplitz(a,b)
t = toeplitz(b)
```

Description

`t = toeplitz(a,b)` returns a nonsymmetric Toeplitz matrix with `a` as its first column and `b` as its first row. `b` is cast to the `numerictype` of `a`. If one of the arguments of `toeplitz` is a built-in data type, it is cast to the data type of the `fi` object. If the first elements of `a` and `b` differ, `toeplitz` issues a warning and uses the column element for the diagonal.

`t = toeplitz(b)` returns the symmetric or Hermitian Toeplitz matrix formed from vector `b`, where `b` is the first row of the matrix.

Examples

Create Symmetric Toeplitz Matrix

```
r = fi([1 2 3]);
toeplitz(r)
```

```
     1     2     3



          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13


        RoundingMethod: Nearest
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision


                   Tag: 

ans =

     1     2     3
     2     1     2
     3     2     1
      numerictype(1,16,13)
```

Create Nonsymmetric Toeplitz Matrix

Create a nonsymmetric Toeplitz matrix with a specified column and row vector.

`toeplitz(a,b)` casts `b` into the data type of `a`. In this example, overflow occurs:

```
fipref('NumericTypeDisplay','short');
format short g
a = fi([1 2 3],true,8,5)
b = fi([1 4 8],true,16,10)
toeplitz(a,b)
```

```
a =

     1     2     3
      numerictype(1,8,5)

b =

     1     4     8
      numerictype(1,16,10)

ans =

            1       3.9688       3.9688
            2            1       3.9688
            3            2            1
      numerictype(1,8,5)
```

`toeplitz(b,a)` casts `a` into the data type of `b`. In this example, overflow does not occur:

```
toeplitz(b,a)
```

```
ans =

     1     2     3
     4     1     2
     8     4     1
      numerictype(1,16,10)
```

If one of the arguments of `toeplitz` is a built-in data type, it is cast to the data type of the `fi` object.

```
x = double([1 exp(1) pi]);
toeplitz(a,x)
```

```
ans =

            1       2.7188       3.1563
            2            1       2.7188
            3            2            1
      numerictype(1,8,5)
```

Input Arguments

a — Column of Toeplitz matrix
scalar | vector

Column of Toeplitz matrix, specified as a scalar or vector. If the first elements of a and b differ, toeplitz uses the column element for the diagonal.

Data Types: fi
Complex Number Support: Yes

b — Row of Toeplitz matrix
scalar | vector

Row of Toeplitz matrix, specified as a scalar or vector. If the first elements of a and b differ, toeplitz uses the column element for the diagonal.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi
Complex Number Support: Yes

Output Arguments

t — Toeplitz matrix
fi object

Toeplitz matrix, returned as a fi object.

The output fi object, t, has the same numerictype properties as the leftmost fi object input. If the leftmost fi object input has a local fimath, the output fi object is assigned the same local fimath. Otherwise, the output fi object, t, has no local fimath.

Version History

Introduced before R2006a

See Also

Blocks
Toeplitz

Functions
toeplitz

tostring

Package: `embedded`

Convert `fi`, `fimath`, `numerictype`, or `quantizer` object to string

Syntax

```
s = tostring(a)
s = tostring(F)
s = tostring(T)
s = tostring(q)
```

Description

`s = tostring(a)` converts `fi` object `a` to a character vector `s` such that `eval(s)` would create a `fi` object with the same properties as `a`.

`s = tostring(F)` converts `fimath` object `F` to a character vector `s` such that `eval(s)` would create a `fimath` object with the same properties as `F`.

`s = tostring(T)` converts `numerictype` object `T` to a character vector `s` such that `eval(s)` would create a `numerictype` object with the same properties as `T`.

`s = tostring(q)` converts `quantizer` object `q` to a character vector `s` such that `eval(s)` would create a `quantizer` object with the same properties as `q`.

Examples

Convert a `fi` Object to a String

```
a = fi(pi,1,16,10);
s = tostring(a)
a1 = eval(s)
isequal(a,a1)
```

```
s =

    'fi('numerictype',numerictype(1,16,10),'Value','3.1416015625')'


a1 =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 10

ans =
```

```
  logical

   1
```

Convert a `fimath` Object to a String

```
F = fimath('OverflowAction','Saturate','RoundingMethod','Convergent');
s = tostring(F)
F1 = eval(s)
isequal(F,F1)
```

```
s =

    'fimath('RoundingMethod', 'Convergent',...
     'OverflowAction', 'Saturate',...
     'ProductMode','FullPrecision',...
     'SumMode','FullPrecision')'


F1 =


        RoundingMethod: Convergent
        OverflowAction: Saturate
           ProductMode: FullPrecision
               SumMode: FullPrecision

ans =

  logical

   1
```

Convert a `numerictype` Object to a String

```
T = numerictype(1,16,15);
s = tostring(T)
T1 = eval(s)
isequal(T,T1)
```

```
s =

    'numerictype(1,16,15)'


T1 =


          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15

ans =
```

```
  logical

   1
```

Convert quantizer Object to a String

```
q = quantizer('fixed','Ceiling','Saturate',[5 4]);
s = tostring(q)
q1 = eval(s)
isequal(q,q1)
```

```
s =

    'quantizer('fixed', 'ceil', 'saturate', [5  4])'


q1 =


        DataMode = fixed
       RoundMode = ceil
    OverflowMode = saturate
          Format = [5  4]

ans =

  logical

   1
```

Input Arguments

a — Input fi object
fi object

Input fi object.

Data Types: fi
Complex Number Support: Yes

F — Input fimath object
fimath object

Input fimath object.

T — Input numerictype object
numerictype object

Input numerictype object.

q — Input quantizer object
quantizer object

Input quantizer object.

Version History

Introduced before R2006a

See Also

`eval` | `fi` | `numerictype` | `fimath` | `quantizer`

ufi

(Not recommended) Construct unsigned fixed-point numeric object

Note `ufi` is not recommended. Use `fi` instead.

Syntax

```
a = ufi
a = ufi(v)
a = ufi(v,w)
a = ufi(v,w,f)
a = ufi(v,w,slope,bias)
a = ufi(v,w,slopeadjustmentfactor,fixedexponent,bias)
```

Description

You can use the `ufi` constructor function in the following ways:

- `a = ufi` is the default constructor and returns an unsigned `fi` object with no value, 16-bit word length, and 15-bit fraction length.
- `a = ufi(v)` returns an unsigned fixed-point object with value `v`, 16-bit word length, and best-precision fraction length.
- `a = ufi(v,w)` returns an unsigned fixed-point object with value `v`, word length `w`, and best-precision fraction length.
- `a = ufi(v,w,f)` returns an unsigned fixed-point object with value `v`, word length `w`, and fraction length `f`.
- `a = ufi(v,w,slope,bias)` returns an unsigned fixed-point object with value `v`, word length `w`, `slope`, and `bias`.
- `a = ufi(v,w,slopeadjustmentfactor,fixedexponent,bias)` returns an unsigned fixed-point object with value `v`, word length `w`, `slopeadjustmentfactor`, `fixedexponent`, and `bias`.

`fi` objects created by the `ufi` constructor function have the following general types of properties:

- fi Object Data Properties
- fimath Object Properties
- numerictype Object Properties

Note `fi` objects created by the `ufi` constructor function have no local fimath.

Examples

Example 1

For example, the following creates an unsigned `fi` object with a value of `pi`, a word length of 8 bits, and a fraction length of 3 bits:

```
a = ufi(pi,8,3)
```

```
a =

    3.1250

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 3
```

Default `fimath` properties are associated with `a`. When a `fi` object does not have a local `fimath` object, no `fimath` object properties are displayed in its output. To determine whether a `fi` object has a local `fimath` object, use the `isfimathlocal` function.

```
isfimathlocal(a)
```

```
ans =
     0
```

A returned value of `0` means the `fi` object does not have a local `fimath` object. When the `isfimathlocal` function returns a `1`, the `fi` object has a local `fimath` object.

Example 2

The value `v` can also be an array:

```
a = ufi((magic(3)/10),16,12)
```

```
a =

    0.8000    0.1001    0.6001
    0.3000    0.5000    0.7000
    0.3999    0.8999    0.2000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 12
```

Example 3

If you omit the argument `f`, it is set automatically to the best precision possible:

```
a = ufi(pi,8)
```

```
a =

    3.1406
```

```
          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 6
```

Example 4

If you omit w and f, they are set automatically to 16 bits and the best precision possible, respectively:

```
a = ufi(pi)
```

```
a = 

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 16
        FractionLength: 14
```

Version History

Introduced in R2009b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- All properties related to data type must be constant for code generation.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

fi | fimath | fipref | isfimathlocal | numerictype | quantizer | sfi

Topics

"View Fixed-Point Data"
"Cast fi Objects"

uint8

Package: `embedded`

Convert `fi` object to unsigned 8-bit integer

Syntax

`c = uint8(a)`

Description

`c = uint8(a)` returns the built-in `uint8` value of `fi` object `a`, based on its real world value. If the data does not fit into an `uint8`, then the data is rounded to nearest and saturated with no warning.

Examples

Find `uint8` Values of `fi` Object

```
a = fi([-pi 0.5 pi],0,8)
```

```
a = 
         0    0.5000    3.1406

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 6
```

```
c = uint8(a)
```

```
c = 1x3 uint8 row vector

   0   1   3
```

Input Arguments

a — Input `fi` object
scalar | vector | matrix | multidimensional array

Input `fi` object, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also
`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint16` | `uint32` | `uint64`

uint16

Convert `fi` object to unsigned 16-bit integer

Syntax

`c = uint16(a)`

Description

`c = uint16(a)` returns the built-in `uint16` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint16`.

Examples

This example shows the `uint16` values of a fi object.

```
a = fi([-pi 0.5 pi],0,16);
c = uint16(a)

c =

   0    1    3
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint32` | `uint64`

uint32

Stored integer value of `fi` object as built-in `uint32`

Syntax

`c = uint32(a)`

Description

`c = uint32(a)` returns the built-in `uint32` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint32`.

Examples

This example shows the `uint32` values of a fi object.

```
a = fi([-pi 0.5 pi],0,32);
c = uint32(a)

c =

   0    1    3
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint64`

uint64

Convert `fi` object to unsigned 64-bit integer

Syntax

`c = uint64(a)`

Description

`c = uint64(a)` returns the built-in `uint64` value of `fi` object `a`, based on its real world value. If necessary, the data is rounded-to-nearest and saturated to fit into an `uint64`.

Examples

This example shows the `uint64` values of a fi object.

```
a = fi([-pi 0.5 pi],0,64);
c = uint64(a)

c =

   0    1    3
```

Version History

Introduced in R2008b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`storedInteger` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

uminus

Negate elements of `fi` object array

Syntax

`uminus(a)`

Description

`uminus(a)` is called for the syntax `-a` when `a` is an object. `-a` negates the elements of `a`.

`uminus` does not support `fi` objects of data type `Boolean`.

Examples

When wrap occurs, -(-1) = -1 :

```
fipref('NumericTypeDisplay','short', ...
       'fimathDisplay','none');
format short g
a = fi(-1,true,8,7,'OverflowMode','wrap')

a =

    -1
      numerictype(1,8,7)

-a

ans =

    -1
      numerictype(1,8,7)

b = fi([-1-i -1-i],true,8,7,'OverflowMode','wrap')

b =

           -1 -           1i             -1 -           1i
      numerictype(1,8,7)

-b

ans =

           -1 -           1i             -1 -           1i
      numerictype(1,8,7)

b'

ans =

           -1 -           1i
```

```
          -1 -           1i
      numerictype(1,8,7)
```

When saturation occurs, -(-1) = 0.99... :

```
c = fi(-1,true,8,7,'OverflowMode','saturate')

c = 

    -1
      numerictype(1,8,7)

-c

ans = 

      0.99219
      numerictype(1,8,7)

d = fi([-1-i -1-i],true,8,7,'OverflowMode','saturate')

d = 

          -1 -           1i           -1 -           1i
      numerictype(1,8,7)

-d

ans = 

      0.99219 +    0.99219i      0.99219 +    0.99219i
      numerictype(1,8,7)

d'

ans = 

          -1 +    0.99219i
          -1 +    0.99219i
      numerictype(1,8,7)
```

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

plus | minus | mtimes | times

unitquantize

Package: `embedded`

Quantize numeric data using `quantizer` object except numbers within `eps` of `+1`

Syntax

```
y = unitquantize(q,x)
[y1,y2,…] = unitquantize(q,x1,x2,…)
```

Description

`y = unitquantize(q,x)` uses the `quantizer` object `q` to quantize numeric data in `x`. `unitquantize` works in the same way as `quantize` except that numbers within `eps(q)` of `+1` are made exactly equal to `+1`.

`[y1,y2,…] = unitquantize(q,x1,x2,…)` is equivalent to `y1 = unitquantize(q,x1), y2 = unitquantize(q,x2), …` and so forth.

Examples

Quantize to Fixed-Point Type

Use `unitquantize` with a `quantizer` object to quantize data.

```
x = (0.8:.1:1.2)';
q = quantizer('fixed','floor','saturate',[4 3]);
y = unitquantize(q,x);
z = [x y]
e = eps(q)
```

```
z =

    0.8000    0.7500
    0.9000    1.0000
    1.0000    1.0000
    1.1000    1.0000
    1.2000    1.0000


e =

    0.1250
```

`unitquantize` quantizes the elements of `x` except for numbers within `eps` of `+1`.

Compare Behavior of `quantize` and `unitquantize`

```
x = [1 pi/4];
q = quantizer([8,7])
```

```
y1 = quantize(q,x)
y2 = unitquantize(q,x)
```

```
q =


        DataMode = fixed
       RoundMode = floor
    OverflowMode = saturate
          Format = [8  7]



Warning: 1 overflow(s) occurred in the fi quantize operation.

y1 =

    0.9922    0.7812


y2 =

    1.0000    0.7812
```

Input Arguments

q — Data type properties
`quantizer` object

Data type properties to use for quantization, specified as a `quantizer` object.

Example: `q = quantizer('fixed','ceil','saturate',[5 4]);`

x — Data to quantize
scalar | vector | matrix | multidimensional array | cell array | structure

Data to quantize, specified as a scalar, vector, matrix, multidimensional array, cell array, or structure.

- When `x` is a numeric array, each element of `x` is quantized.
- When `x` is a cell array, each numeric element of the cell array is quantized.
- When `x` is a structure, each numeric field of `x` is quantized.

`unitquantize` does not change nonnumeric elements or fields of `x`, nor does it issue warnings for nonnumeric values. Numbers within `eps(q)` of `+1` are made exactly equal to `+1`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `struct` | `cell`
Complex Number Support: Yes

`x1,x2,…` — Data to quantize (as separate elements)
scalar | vector | matrix | multidimensional array | cell array | structure

Data to quantize (as separate elements), specified as a scalar, vector, matrix, multidimensional array, cell array, or structure.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `struct` | `cell`
Complex Number Support: Yes

Version History

Introduced in R2008a

See Also

`eps` | `quantize` | `quantizer`

unitquantizer

Create `unitquantizer` object

Description

The `unitquantizer` object describes data type properties to use for quantization. After you create a `unitquantizer` object, use `quantize` to quantize double-precision data. A `unitquantizer` object is the same as a `quantizer` object except that its `quantize` method quantizes numbers within `eps(q)` of `+1` to exactly `+1`. You can use the `unitquantizer` object to simulate custom floating-point data types with arbitrary word length and exponent length.

Creation

Syntax

```
q = unitquantizer
q = unitquantizer(Name,Value)
q = unitquantizer(Value1,Value2)
q = unitquantizer(s)
q = unitquantizer(pn,pv)
```

Description

`q = unitquantizer` creates a `unitquantizer` object with properties set to their default values. To use this object to quantize values, use `quantize`.

`q = unitquantizer(Name,Value)` sets named properties using name-value arguments. You can specify multiple name-value arguments. Enclose each property name in single quotes.

`q = unitquantizer(Value1,Value2)` sets properties using property values. Property values are unique, so you can set property names by specifying just the property values in the command. When two values conflict, `unitquantizer` sets the last property value in the list.

`q = unitquantizer(s)` sets properties named in each field name with the values contained in the structure `s`.

`q = unitquantizer(pn,pv)` sets the named properties specified in the cell array of character vectors `pn` to the corresponding values in the cell array `pv`.

You can use a combination of name-value string arguments, structures, and name-value cell array arguments to set property values when creating a `unitquantizer` object.

Properties

`DataMode` — Data type mode
`'fixed'` (default) | `'ufixed'` | `'float'` | `'single'` | `'double'`

Type of arithmetic used in quantization, specified as one of these values:

- `'fixed'` — Signed fixed-point mode.
- `'ufixed'` — Unsigned fixed-point mode.
- `'float'` — Custom-precision floating-point mode.
- `'single'` — Single-precision mode. This mode overrides all other property settings.
- `'double'` — Double-precision mode. This mode overrides all other property settings.

Data Types: `char` | `struct` | `cell`

RoundMode — Rounding method
`'floor'` (default) | `'ceil'` | `'convergent'` | `'fix'` | `'nearest'` | `'round'`

Rounding method to use, specified as one of these values:

- `'ceil'` — Round up to the next allowable quantized value.
- `'convergent'` — Round to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit after rounding would be set to 0.
- `'fix'` — Round negative numbers up and positive numbers down to the next allowable quantized value.
- `'floor'` — Round down to the next allowable quantized value.
- `'nearest'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up.
- `'round'` — Round to the nearest allowable quantized value. Numbers that are halfway between the two nearest allowable quantized values are rounded up in absolute value.

Data Types: `char` | `struct` | `cell`

`OverflowMode` — Action to take on overflow
`'saturate'` (default) | `'wrap'`

Action to take on overflow, specified as one of these values:

- `'saturate'` — Overflows saturate.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers as specified by the data format properties, these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.
- `'wrap'` — Overflows wrap to the range of representable values.

  When the values of data to be quantized lie outside the range of the largest and smallest representable numbers as specified by the data format properties, these values are wrapped back into that range using modular arithmetic relative to the smallest representable number.

This property only applies to fixed-point data type modes. This property becomes a read-only property when you set the `DataMode` property to `float`, `double`, or `single`.

Note Floating-point numbers that extend beyond the dynamic range overflow to `±Inf`.

Data Types: `char` | `struct` | `cell`

Format — Data format of unitquantizer object
[16 15] (default) | [wordlength fractionlength] | [wordlength exponenetlength] | [64 11] | [32 8]

Data format of unitquantizer object. The interpretation of this property value depends on the value of the DataMode property.

DataMode Property Value	Interpreting the Format Property Values
fixed or ufixed	[wordlength fractionlength] Specify the Format property value as a two-element row vector where the first element is the number of bits for the quantizer object word length and the second element is the number of bits for the quantizer object fraction length. The word length can range from 2 to the limits of memory on your PC. The fraction length can range from 0 to one less than the word length.
float	[wordlength exponenetlength] Specify the Format property value as a two-element row vector where the first element is the number of bits for the unitquantizer object word length and the second element is the number of bits for the unitquantizer object exponent length. The word length can range from 2 to the limits of memory on your PC. The fraction length can range from 0 to 11.
double	[64 11] The read-only Format property value automatically specifies the word length and exponent length.
single	[32 8] The read-only Format property value automatically specifies the word length and exponent length.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Read-Only unitquantizer Object States

Read-only unitquantizer object states are updated when quantize is called. To reset these states, use reset.

max — Maximum value before quantization
scalar

Maximum value before quantization during a call to `quantize(q,…)` for `unitquantizer` object `q`, specified as a scalar. This value is the maximum value recorded over successive calls to `quantize`.

Example: `max(q)`

Example: `q.max`

`min` — Minimum value before quantization
scalar

Minimum value before quantization during a call to `quantize(q,…)` for `unitquantizer` object `q`, specified as a scalar. This value is the minimum value recorded over successive calls to `quantize`.

Example: `min(q)`

Example: `q.min`

`noverflows` — Number of overflows
scalar

Number of overflows during a call to `quantize(q,…)` for `unitquantizer` object `q`, specified as a scalar. This value accumulates over successive calls to `quantize`. An overflow is defined as a value that when quantized is outside the range of `q`.

Example: `noverflows(q)`

Example: `q.noverflows`

`nunderflows` — Number of underflows
scalar

Number of underflows during a call to `quantize(q,…)` for `unitquantizer` object `q`. This value accumulates over successive calls to `quantize`. An underflow is defined as a number that is nonzero before it is quantized and zero after it is quantized.

Example: `nunderflows(q)`

Example: `q.nunderflows`

`noperations` — Number of data points quantized
scalar

Number of quantization operations during a call to `quantize(q,…)` for `unitquantizer` object `q`. This value accumulates over successive calls to `quantize`.

Example: `noperations(q)`

Example: `q.noperations`

Object Functions

Examples

Quantize Data with `unitquantizer` Object

Quantize a vector `x` using the `unitquantizer` object `q`.

```
x = (0.8:.1:1.2)';
q = unitquantizer([4 3]);
y = quantize(q,x);
z = [x y]
e = eps(q)

z =

    0.8000    0.7500
    0.9000    1.0000
    1.0000    1.0000
    1.1000    1.0000
    1.2000    1.0000


e =

    0.1250
```

`quantize` quantizes the elements of `x` except for numbers within `eps` of `+1`.

Version History

Introduced in R2008a

See Also

`quantize` | `quantizer` | `unitquantize` | `assignmentquantizer` | `reset`

unshiftdata

Inverse of `shiftdata`

Syntax

`y = unshiftdata(x,perm,nshifts)`

Description

`y = unshiftdata(x,perm,nshifts)` restores the orientation of the data that was shifted with `shiftdata`. The permutation vector is given by `perm`, and `nshifts` is the number of shifts that was returned from `shiftdata`.

`unshiftdata` is meant to be used in tandem with `shiftdata`. These functions are useful for creating functions that work along a certain dimension, like `filter`, `goertzel`, `sgolayfilt`, and `sosfilt`.

Examples

Example 1

1 Create a `3-by-3` magic square:

```
x = fi(magic(3))
```

```
x =

     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

2 Shift the matrix `x` to work along the second dimension:

```
[x,perm,nshifts] = shiftdata(x,2)
```

```
x =

     8     3     4
     1     5     9
     6     7     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11

perm =
```

```
     2     1


nshifts =

     []
```

This command returns the permutation vector, `perm`, and the number of shifts, `nshifts`, are returned along with the shifted matrix, `x`.

3 Shift the matrix back to its original shape:

```
y = unshiftdata(x,perm,nshifts)

y =

     8     1     6
     3     5     7
     4     9     2

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 11
```

Example 2

1 Define `x` as a row vector:

```
x = 1:5

x =

     1     2     3     4     5
```

2 Define `dim` as empty to shift the first non-singleton dimension of `x` to the first column:

```
[x,perm,nshifts] = shiftdata(x,[])

x =

     1
     2
     3
     4
     5


perm =

     []


nshifts =

     1
```

This command returns `x` as a column vector, along with `perm`, the permutation vector, and `nshifts`, the number of shifts.

3 Using `unshiftdata`, restore `x` to its original shape:

```
y = unshiftdata(x,perm,nshifts)
```

```
y =

     1     2     3     4     5
```

Version History

Introduced in R2008a

See Also

`shiftdata`

upperbound

Upper bound of range of `fi` object

Syntax

```
u = upperbound(a)
```

Description

`u = upperbound(a)` returns the upper bound of the range of `fi` object `a`.

If `l = lowerbound(a)` and `u = upperbound(a)`, then `[l,u] = range(a)`.

Examples

Upper Bound of fi Object

```
a = fi(pi,1,16,3,2)
```

```
a = 
     2

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

```
u = upperbound(a)
```

```
u = 
       98303

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 16
                 Slope: 3
                  Bias: 2
```

Input Arguments

a — Input `fi` object
`fi` object

Input `fi` object.

Data Types: `fi`

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`eps` | `fi` | `intmax` | `intmin` | `lowerbound` | `lsb` | `range` | `realmax` | `realmin`

vertcat

Package: embedded

Concatenate fi object arrays vertically

Syntax

```
C = vertcat(A,B)
C = vertcat(A1,A2,…An)
```

Description

C = vertcat(A,B) concatenates B vertically to the end of A when any of A and B is a fi object.

A and B must have the same number of columns. Multidimensional arrays are vertically concatenated along the first dimension. The remaining dimensions must match.

C = vertcat(A1,A2,…An) concatenates A1,A2,…An vertically when any of A1,A2,…An is a fi object.

A and B must have the same number of columns. Multidimensional arrays are vertically concatenated along the first dimension. The remaining dimensions must match.

vertcat is equivalent to using square brackets for vertically concatenating arrays. For example, [A; B] is equal to vertcat(A,B) when A and B are compatible arrays.

Horizontal and vertical concatenation can be combined, as in [a b;c d].

[a b; c] is allowed if the number of rows of a equals the number of rows of b, and if the number of columns of a plus the number of columns of b equals the number of columns of c.

The matrices in a concatenation expression can themselves be formed via a concatenation, as in [a b;[c d]].

Note The fimath and numerictype objects of a concatenated matrix of fi objects C are taken from the leftmost fi object in the list A1,A2,…An.

Examples

Concatenate Two Matrices

Create two matrices and concatenate them vertically, first by using square bracket notation, and then by using vertcat.

```
A = fi([1 2 3; 4 5 6])
```

```
A =

     1     2     3
```

```
     4     5     6

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

```
B = fi([7 8 9],0,8)
```

```
B =

     7     8     9

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
            WordLength: 8
        FractionLength: 4
```

```
C = [A; B]
```

```
C =

    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    7.9998    7.9998

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

```
D = vertcat(A,B)
```

```
D =

    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    7.9998    7.9998

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 12
```

Note that the `numerictype` of concatenated matrix `D` is taken from the leftmost `fi` object in the input list.

Input Arguments

A — First input
scalar | vector | matrix | multidimensional array

First input, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

B — Second input
scalar | vector | matrix | multidimensional array

Second input, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

A1,A2,…An — List of inputs
comma-separated list

List of inputs, specified as a comma-separated list of elements to concatenate in the order they are specified.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`
Complex Number Support: Yes

Version History

Introduced before R2006a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

See Also

`horzcat` | `fi` | `fimath` | `numerictype`

wordlength

Package: `embedded`

Word length of `quantizer` object

Syntax

`wl = wordlength(q)`

Description

`wl = wordlength(q)` returns the word length in bits of quantizer object `q`.

Examples

Word Length of `quantizer` Object

```
q = quantizer([16 15]);
wordlength(q)
```

```
ans = 16
```

The word length is the first element of `format(q)`.

```
format(q)
```

```
ans = 1×2

    16    15
```

Input Arguments

q — `quantizer` object
`quantizer` object

`quantizer` object to find word length of.

Version History

Introduced before R2006a

See Also

`fi` | `fractionlength` | `exponentlength` | `numerictype` | `quantizer`

zeros

Create array of all zeros with fixed-point properties

Syntax

```
X = zeros('like',p)
X = zeros(n,'like',p)
X = zeros(sz1,...,szN,'like',p)
X = zeros(sz,'like',p)
```

Description

`X = zeros('like',p)` returns a scalar `0` with the same `numerictype`, complexity (real or complex), and `fimath` as `p`.

`X = zeros(n,'like',p)` returns an `n`-by-`n` array of zeros like `p`.

`X = zeros(sz1,...,szN,'like',p)` returns an `sz1`-by-...-by-`szN` array of zeros like `p`.

`X = zeros(sz,'like',p)` returns an array of zeros like `p`. The size vector, `sz`, defines `size(X)`.

Examples

2-D Array of Zeros With Fixed-Point Attributes

Create a 2-by-3 array of zeros with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 2-by-3 array of zeros that has the same numerictype properties as `p`.

```
X = zeros(2,3,'like',p)
```

```
X =
     0     0     0
     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Size Defined by Existing Array

Define a 3-by-2 array `A`.

```
A = [1 4 ; 2 5 ; 3 6];
```

```
sz = size(A)
```

```
sz = 1×2

     3     2
```

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create an array of zeros that is the same size as `A` and has the same numerictype properties as `p`.

```
X = zeros(sz,'like',p)
```

```
X = 
     0     0
     0     0
     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Square Array of Zeros With Fixed-Point Attributes

Create a 4-by-4 array of zeros with specified numerictype and `fimath` properties.

Create a signed `fi` object with word length of `24` and fraction length of `12`.

```
p = fi([],1,24,12);
```

Create a 4-by-4 array of zeros that has the same numerictype properties as `p`.

```
X = zeros(4, 'like', p)
```

```
X = 
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Complex Fixed-Point Zero

Create a scalar fixed-point `0` that is not real valued, but instead is complex like an existing array.

Define a complex `fi` object.

```
p = fi( [1+2i 3i],1,24,12);
```

Create a scalar `1` that is complex like `p`.

```
X = zeros('like',p)
```

```
X =
   0.0000 + 0.0000i

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 24
        FractionLength: 12
```

Write MATLAB Code That Is Independent of Data Types

Write a MATLAB algorithm that you can run with different data types without changing the algorithm itself. To reuse the algorithm, define the data types separately from the algorithm.

This approach allows you to define a baseline by running the algorithm with floating-point data types. You can then test the algorithm with different fixed-point data types and compare the fixed-point behavior to the baseline without making any modifications to the original MATLAB code.

Write a MATLAB function, `my_filter`, that takes an input parameter, `T`, which is a structure that defines the data types of the coefficients and the input and output data.

```
function [y,z] = my_filter(b,a,x,z,T)
    % Cast the coefficients to the coefficient type
    b = cast(b,'like',T.coeffs);
    a = cast(a,'like',T.coeffs);
    % Create the output using zeros with the data type
    y = zeros(size(x),'like',T.data);
    for i = 1:length(x)
        y(i) = b(1)*x(i) + z(1);
        z(1) = b(2)*x(i) + z(2) - a(2) * y(i);
        z(2) = b(3)*x(i)        - a(3) * y(i);
    end
end
```

Write a MATLAB function, `zeros_ones_cast_example`, that calls `my_filter` with a floating-point step input and a fixed-point step input, and then compares the results.

```
function zeros_ones_cast_example

    % Define coefficients for a filter with specification
    % [b,a] = butter(2,0.25)
    b = [0.097631072937818   0.195262145875635   0.097631072937818];
    a = [1.000000000000000  -0.942809041582063   0.333333333333333];

    % Define floating-point types
    T_float.coeffs = double([]);
    T_float.data   = double([]);
```

```
    % Create a step input using ones with the
    % floating-point data type
    t = 0:20;
    x_float = ones(size(t),'like',T_float.data);

    % Initialize the states using zeros with the
    % floating-point data type
    z_float = zeros(1,2,'like',T_float.data);

    % Run the floating-point algorithm
    y_float = my_filter(b,a,x_float,z_float,T_float);

    % Define fixed-point types
    T_fixed.coeffs = fi([],true,8,6);
    T_fixed.data   = fi([],true,8,6);

    % Create a step input using ones with the
    % fixed-point data type
    x_fixed = ones(size(t),'like',T_fixed.data);

    % Initialize the states using zeros with the
    % fixed-point data type
    z_fixed = zeros(1,2,'like',T_fixed.data);

    % Run the fixed-point algorithm
    y_fixed = my_filter(b,a,x_fixed,z_fixed,T_fixed);

    % Compare the results
    coder.extrinsic('clf','subplot','plot','legend')
    clf
    subplot(211)
    plot(t,y_float,'co-',t,y_fixed,'kx-')
    legend('Floating-point output','Fixed-point output')
    title('Step response')
    subplot(212)
    plot(t,y_float - double(y_fixed),'rs-')
    legend('Error')
    figure(gcf)
end
```

Input Arguments

n — Size of square matrix

integer value

Size of square matrix, specified as an integer value, defines the output as a square, n-by-n matrix of ones.

- If n is zero, X is an empty matrix.
- If n is negative, it is treated as zero.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz1,...,szN — Size of each dimension

two or more integer values

Size of each dimension, specified as two or more integer values, defines X as a sz1-by...-by-szN array.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

sz — Output size
row vector of integer values

Output size, specified as a row vector of integer values. Each element of this vector indicates the size of the corresponding dimension.

- If the size of any dimension is zero, X is an empty array.
- If the size of any dimension is negative, it is treated as zero.
- If any trailing dimensions greater than two have a size of one, the output, X, does not include those dimensions.

Example: `sz = [2,3,4]` defines X as a 2-by-3-by-4 array.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

p — Prototype
fi object | numeric variable

Prototype, specified as a `fi` object or numeric variable. To use the prototype to specify a complex object, you must specify a value for the prototype. Otherwise, you do not need to specify a value.

Complex Number Support: Yes

Tips

Using the `b = cast(a,'like',p)` syntax to specify data types separately from algorithm code allows you to:

- Reuse your algorithm code with different data types.
- Keep your algorithm uncluttered with data type specifications and switch statements for different data types.
- Improve readability of your algorithm code.
- Switch between fixed-point and floating-point data types to compare baselines.
- Switch between variations of fixed-point settings without changing the algorithm code.

Version History

Introduced in R2013a

See Also

`cast` | `ones` | `zeros`

Topics
"Implement FIR Filter Algorithm for Floating-Point and Fixed-Point Types Using cast and zeros"

“Manual Fixed-Point Conversion Workflow”
“Manual Fixed-Point Conversion Best Practices”

5

Classes

coder.CellType class

Package: coder
Superclasses: coder.ArrayType

Represent set of MATLAB cell arrays

Description

Specifies the set of cell arrays that the generated code accepts. Use only with the `fiaccel -args` option. Do not pass as an input to a generated MEX function.

Construction

Note You can also create and edit `coder.Type` objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

`t = coder.typeof(cells)` creates a `coder.CellType` object for a cell array that has the same cells and cell types as `cells`. The cells in `cells` are type objects or example values.

`t = coder.typeof(cells,sz,variable_dims)` creates a `coder.CellType` object that has upper bounds specified by `sz` and variable dimensions specified by `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds do not change. If you do not specify the `variable_dims` input parameter, except for the unbounded dimensions, the dimensions of the type are fixed. A scalar `variable_dims` applies to the bounded dimensions that are not `1` or `0`.

When `cells` specifies a cell array whose elements have different classes, you cannot use `coder.typeof` to create a `coder.CellType` object for a variable-size cell array.

`t = coder.newtype('cell',cells)` creates a `coder.CellType` object for a cell array that has the cells and cell types specified by `cells`. The cells in `cells` must be type objects.

`t = coder.newtype('cell',cells,sz,variable_dims)` creates a `coder.CellType` that has upper bounds specified by `sz` and variable dimensions specified by `variable_dims`. If `sz` specifies `inf` for a dimension, then the size of the dimension is unbounded and the dimension is variable size. When `sz` is `[]`, the upper bounds do not change. If you do not specify the `variable_dims` input parameter, except for the unbounded dimensions, the dimensions of the type are fixed. A scalar `variable_dims` applies to the bounded dimensions that are not `1` or `0`.

When `cells` specifies a cell array whose elements have different classes, you cannot use `coder.newtype` to create a `coder.CellType` object for a variable-size cell array.

Input Arguments

`cells` — Specification of cell types
cell array

Cell array that specifies the cells and cell types for the output `coder.CellType` object. For `coder.typeof`, `cells` can contain type objects or example values. For `coder.newtype`, `cells` must contain type objects.

`sz` — Size of cell array
row vector of integer values

Specifies the upper bound for each dimension of the cell array type object. For `coder.newtype`, `sz` cannot change the number of cells for a heterogeneous cell array.

For `coder.newtype`, the default is `[1 1]`.

`variable_dims` — Dimensions that are variable size
row vector of logical values

Specifies whether each dimension is variable size (true) or fixed size (false).

For `coder.newtype`, the default is `true` for dimensions for which `sz` specifies an upper bound of `inf` and `false` for all other dimensions.

When `cells` specifies a cell array whose elements have different classes, you cannot create a `coder.CellType` object for a variable-size cell array.

Properties

`Cells` — Types of cells
cell array

A cell array that specifies the `coder.Type` of each cell.

`ClassName` — Name of class
character vector or string scalar

Class of values in this set.

`SizeVector` — Size of cell array
row vector of integer values

The upper bounds of dimensions of the cell array.

`VariableDims` — Dimensions that are variable size
row vector of logical values

A vector that specifies whether each dimension of the array is fixed or variable size. If a vector element is `true`, the corresponding dimension is variable size.

Methods

isHeterogeneous	Determine whether cell array type represents a heterogeneous cell array
isHomogeneous	Determine whether cell array type represents a homogeneous cell array
makeHeterogeneous	Make a heterogeneous copy of a cell array type
makeHomogeneous	Create a homogeneous copy of a cell array type

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Examples

Create a Type for a Cell Array Whose Elements Have the Same Class

Create a type for a cell array whose first element has class char and whose second element has class double.

```
t = coder.typeof({1 2 3})
```

```
t = 

coder.CellType
   1x3 homogeneous cell 
      base: 1x1 double
```

The type is homogeneous.

Create a Heterogeneous Type for a Cell Array Whose Elements Have the Same Class

To create a heterogeneous type when the elements of the example cell array type have the same class, use the `makeHeterogeneous` method.

```
t = makeHeterogeneous(coder.typeof({1 2 3}))
```

```
t = 

coder.CellType
   1×3 locked heterogeneous cell 
      f1: 1×1 double
      f2: 1×1 double
      f3: 1×1 double
```

The cell array type is heterogeneous. It is represented as a structure in the generated code.

Create a Cell Array Type for a Cell Array Whose Elements Have Different Classes

Define variables that are example cell values.

```
a = 'a';
b = 1;
```

Pass the example cell values to `coder.typeof`.

```
t = coder.typeof({a, b})
```

```
t = 

coder.CellType
```

```
   1x2 heterogeneous cell
      f0: 1x1 char
      f1: 1x1 double
```

Create a Type for a Variable-Size Homogeneous Cell Array from an Example Cell Array Whose Elements Have Different Classes

Create a type for a cell array that contains two character vectors that have different sizes.

```
t = coder.typeof({'aa', 'bbb'})
```

```
t = 

coder.CellType
   1x2 heterogeneous cell
      f0: 1x2 char
      f1: 1x3 char
```

The cell array type is heterogeneous.

Create a type using the same cell array input. This time, specify that the cell array type has variable-size dimensions.

```
t = coder.typeof({'aa','bbb'},[1,10],[0,1])
```

```
t = 

coder.CellType
   1×:10 locked homogeneous cell
      base: 1×:3 char
```

The cell array type is homogeneous. `coder.typeof` determined that the base type `1x:3 char` can represent `'aa'`, and `'bbb'`.

Create a New Cell Array Type from a Cell Array of Types

Create a type for a scalar `int8`.

```
ta = coder.newtype('int8',[1 1]);
```

Create a type for a :1x:2 double row vector.

```
tb = coder.newtype('double',[1 2],[1 1]);
```

Create a cell array type whose cells have the types specified by `ta` and `ta`.

```
t = coder.newtype('cell',{ta,tb})
```

```
t = 

coder.CellType
   1x2 heterogeneous cell
```

```
f0: 1x1 int8
f1: :1x:2 double
```

Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

  The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

Version History

Introduced in R2015b

See Also

`coder.ClassType` | `coder.ArrayType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof` | `fiaccel`

Topics

"Code Generation for Cell Arrays"
"Create and Edit Input Types by Using the Coder Type Editor"

coder.ClassType class

Package: `coder`
Superclasses: `coder.ArrayType`

Represent set of MATLAB classes acceptable for input specification

Description

Objects of the `coder.ClassType` specify value class objects that the generated code accepts. Use objects of this class only with the `-args` option of the `fiaccel` function. Do not pass as an input to a generated MEX function.

Creation

`t = coder.typeof(classObject)` creates a `coder.ClassType` object for `classObject`.

`t = coder.newtype(className)` creates a `coder.ClassType` object for an object of the `className` class.

Note You can create and edit `coder.Type` objects interactively by using the Coder Type Editor. See "Create and Edit Input Types by Using the Coder Type Editor".

Input Arguments

`classObject` — Value class object
MATLAB class

Value class object for which to create the `coder.ClassType` object. This input is an expression that evaluates to an object of a value class.

`className` — Name of value class definition
string scalar | character vector

Name of a value class definition file on the MATLAB path specified as a character vector or string scalar.

Properties

When you create a `coder.ClassType` object `t` by passing a value class object `v` to `coder.typeof`, `t` has same as the properties as `v` with the `Constant` attribute set to `false`.

Similarly, when you create a `coder.ClassType` object `t` by passing the name of the value class object, `v` to `coder.newtype`, `t` has same as the properties as `v` with the `Constant` attribute set to `false`.

Examples

Create Type-Based Example Object

This example shows how to create a type object based on an example object in the workspace.

Create a value class `myRectangle`.

```
type myRectangle.m
```

```
classdef myRectangle
    properties
        length;
        width;
    end
    methods
        function obj = myRectangle(l,w)
            if nargin > 0
                obj.length = l;
                obj.width = w;
            end
        end
        function area = calcarea(obj)
            area = obj.length * obj.width;
        end
    end
end
```

Create a function that takes an object of `myRectangle` as the input.

```
type getarea.m
```

```
function z = getarea(r)
%#codegen
z = calcarea(r);
end
```

Create an object of `myRectangle`.

```
v = myRectangle(1,2)
```

```
v = 
  myRectangle with properties:

    length: 1
     width: 2
```

Create a `coder.ClassType` object based on `v`.

```
t = coder.typeof(v)
```

```
t = 
coder.ClassType
   1×1 myRectangle
      Properties :
          length : 1×1 double
          width  : 1×1 double

    Edit Type Object
```

`coder.typeof` creates a `coder.ClassType` object that has the same properties names and types as `v`.

Generate code for `getarea`. Specify the input type by passing the `coder.ClassType` object, `t`, to the `-args` option.

```
codegen getarea -args {t} -report
```

```
Code generation successful: To view the report, open('codegen\mex\getarea\html\report.mldatx')
```

Create Type by Using `coder.newtype`

This example shows how to create a `coder.ClassType` object for an object of the value class `mySquare` by using `coder.newtype`.

Create a value class `mySquare` that has one property, `side`.

```
type mySquare.m
```

```
classdef mySquare
    properties
        side;
    end
    methods
        function obj = mySquare(val)
            if nargin > 0
                obj.side = val;
            end
        end
        function a = calcarea(obj)
            a = obj.side * obj.side;
        end
    end
end
```

Create a `coder.ClassType` type for `mySquare` without assiging any property values.

```
t = coder.newtype('mySquare')
```

```
t = 
coder.ClassType
   1×1 mySquare -- class with no properties

    Edit Type Object
```

To ensure that `t` has the properties of `mySquare`, specify the type of `side` by using `t.Properties`.

```
t.Properties.side = coder.typeof(2)
```

```
t = 
coder.ClassType
   1×1 mySquare
      Properties :
          side : 1×1 double

    Edit Type Object
```

Tips

- After you create a `coder.ClassType`, you can modify the types of the properties. For example, modify the type of the `prop1` and `prop2` properties of an object `t`:

  ```
  t = coder.typeof(myClass)
  t.Properties.prop1 = coder.typeof(int16(2));
  t.Properties.prop2 = coder.typeof([1 2 3]);
  ```
- After you create a `coder.ClassType` object, you can add properties. For example, add the `newprop1` and `newprop2` properties of an object `t`:

  ```
  t = coder.typeof(myClass)
  t.Properties.newprop1 = coder.typeof(int8(2));
  t.Properties.newprop2 = coder.typeof([1 2 3]);
  ```
- When you generate code, the properties of the `coder.ClassType` object that you pass to the `codegen` function must be consistent with the properties in the class definition file. However, if the class definition file has properties that your code does not use, the `coder.ClassType` object does not have to include those properties. The code generator ignores properties that your code does not use.

Version History

Introduced in R2017a

See Also

`coder.CellType` | `coder.Type` | `coder.PrimitiveType` | `coder.EnumType` | `coder.CellType` | `coder.FiType` | `coder.Constant` | `coder.ArrayType` | `coder.newtype` | `coder.typeof` | `coder.resize` | `fiaccel`

Topics

"Create and Edit Input Types by Using the Coder Type Editor"

coder.mexconfig

Code acceleration configuration object for use with `fiaccel`

Description

A `coder.MexConfig` object contains all the configuration parameters that the `fiaccel` function uses when accelerating fixed-point code via a generated MEX function. To use this object, first create it using the `coder.mexconfig` function, then pass it to the `fiaccel` function using the `-config` option.

Creation

Syntax

`cfg = coder.mexconfig`

Description

`cfg = coder.mexconfig` creates a `coder.MexConfig` code generation object for use with `fiaccel`, which generates a MEX function.

Properties

`CompileTimeRecursionLimit` — Number of copies of a function allowed in generated code for compile-time recursion
50 (default) | integer

Number of copies of a function allowed in generated code for compile-time recursion, specified as an integer. To disallow recursion in the MATLAB code, set `CompileTimeRecursionLimit` to `0`. The default compile-time recursion limit is high enough for most recursive functions that require compile-time recursion. If code generation fails because of the compile-time recursion limit, and you want compile-time recursion, try to increase the limit. Alternatively, change your MATLAB code so that the code generator uses run-time recursion.

`ConstantFoldingTimeout` — Maximum number of constant folder instructions
10000 (default) | positive integer

Maximum number of instructions to be executed by the constant folder, specified as a positive integer.

`ConstantInputs` — Constant input checking mode
`'CheckValues'` (default) | `'IgnoreValues'` | `'Remove'`

Constant input checking mode, specified as one of the values in this table.

Value	Description
'CheckValues'	This value is the default value. When you call the MEX function, it checks that the value you provide for a constant input argument is the value specified at code generation time. You can call the MEX function and the original MATLAB function with the same arguments. Therefore, you can use the same test file for both functions. Checking the values can slow down execution of the MEX function.
'IgnoreValues'	When you call the MEX function, it ignores the value that you provide for a constant input argument. It uses the value specified at code generation time. You can use the same test file without the overhead of checking the constant argument values.
'Remove'	The code generator removes constant input arguments from the MEX function signature. When you call the MEX function, you do not provide a value for a constant input argument. This option provides backward compatibility.

See "Constant Input Checking in MEX Functions" (MATLAB Coder).

`DynamicMemoryAllocation` — Dynamic memory allocation for variable-size data
`'Threshold'` (default) | `'AllVariableSizeArrays'` | `'Off'`

Dynamic memory allocation for variable-size data, specified as one of these values:

- `'Threshold'` — Dynamic memory allocation is enabled for all variable-size arrays whose size is greater than `DynamicMemoryAllocationThreshold` and `fiaccel` allocates memory for this variable-size data dynamically on the heap.
- `'AllVariableSizeArrays'` — Allocate memory for all variable-size arrays dynamically on the heap. You must use dynamic memory allocation for all unbounded variable-size data.
- `'Off'` — Allocate memory statically on the stack.

You must use dynamic memory allocation for all unbounded variable-size data.

Dependencies:

- `EnableVariableSizing` enables this parameter.
- Setting this `DynamicMemoryAllocation` to `'Threshold'` enables the `DynamicMemoryAllocationThreshold` parameter.

Note `DynamicMemoryAllocation` configuration option will be removed in a future release. To dynamically allocation memory for variable-sized arrays, use the `EnableDynamicMemoryAllocation` option. To set the threshold, use the `DynamicMemoryAllocationThreshold` option.

`DynamicMemoryAllocationForFixedSizeArrays` — Dynamic memory allocation for fixed-size arrays
`false` (default) | `true`

Dynamic memory allocation for fixed-size arrays, specified as one of the values in this table.

Value	Description
`true`	The code generator allocates memory dynamically on the heap for fixed-size arrays whose size (in bytes) is greater than or equal to `DynamicMemoryAllocationThreshold`.
`false`	This value is the default value. The code generator statically allocates memory for fixed-size arrays on the stack.

Dependency:

- Setting `EnableDynamicMemoryAllocation` to `true` enables this option.

See, "Control Dynamic Memory Allocation for Fixed-Size Arrays" (MATLAB Coder).

`DynamicMemoryAllocationThreshold` — Memory allocation threshold
`65536` (default) | integer

Size of the threshold for variable-size arrays above which `fiaccel` allocates memory on the heap, specified as an integer.

Dependency:

- Setting `EnableDynamicMemoryAllocation` to `true` enables this parameter.

`EnableAutoExtrinsicCalls` — Treat common visualization functions as extrinsic functions
`true` (default) | `false`

Whether `fiaccel` treats common visualization functions as extrinsic functions, specified as `true` or `false`. When this option is enabled, `fiaccel` detects calls to many common visualization functions, such as `plot`, `disp`, and `figure`. It calls out to MATLAB for these functions. This capability reduces the amount of time that you spend making your code suitable for code generation. It also removes the requirement to declare these functions extrinsic using the `coder.extrinsic` function.

`EchoExpressions` — Show results of code not terminated with semicolons
`true` (default) | `false`

Whether to show results of code not terminated with semicolons, specified as `true` or `false`.

Set this property to `true` to have the results of code instructions that do not terminate with a semicolon appear in the MATLAB Command Window. If you set this property to `false`, code results do not appear in the MATLAB Command Window.

`EnableRuntimeRecursion` — Allow recursive functions in the generated code
`true` (default) | `false`

Whether to allow recursive functions in the generated code, specified as `true` or `false`. If your MATLAB code requires run-time recursion and this parameter is `false`, code generation fails.

`EnableDebugging` — Compile generated code in debug mode
`false` (default) | `true`

Whether to compile generated code in debug mode, specified as `true` or `false`.

Set this property to `true` to compile the generated code in debug mode. Set this property to `false` to compile the code in normal mode.

`EnableDynamicMemoryAllocation` — Enable dynamic memory allocation for variable-size arrays
`true` (default) | `false`

Dynamic memory allocation for variable-size arrays, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. The code generator allocates memory dynamically on the heap for variable-size arrays whose size (in bytes) is greater than or equal to `DynamicMemoryAllocationThreshold`.
`false`	The code generator statically allocates memory for variable-size arrays on the stack.

`EnableImplicitExpansion` — Enable implicit expansion capabilities in generated code
`true` (default) | `false`

Whether to enable implicit expansion capabilities in generated code, specified as `true` or `false`.

Set this property to `true` to enable implicit expansion in the generated code. The code generator includes modifications in the generated code to apply implicit expansion. See “Compatible Array Sizes for Basic Operations”. Set this property to `false` so the generated code does not follow the rules of implicit expansion.

`EnableVariableSizing` — Enable support for variable-sized arrays
`true` (default) | `false`

Whether to enable support for variable-sized arrays, specified as `true` or `false`.

Set this property to `true` to enable support for variable-sized arrays and to enable the `DynamicMemoryAllocation` property. If you set this property to `false`, variable-sized arrays are not supported.

`ExtrinsicCalls` — Extrinsic function calls
`true` (default) | `false`

An extrinsic function is a function on the MATLAB path that the generated code dispatches to MATLAB software for execution. `fiaccel` does not compile or generate code for extrinsic functions.

Set this property to `true` to have `fiaccel` generate code for the call to a MATLAB function, but not generate the function's internal code. Set this property to `false` to have `fiaccel` ignore the extrinsic function and not generate code for the call to the MATLAB function. If the extrinsic function affects the output of `fiaccel`, a compiler error occurs.

`ExtrinsicCalls` affects how MEX functions built by `fiaccel` generate random numbers when using the MATLAB `rand`, `randi`, and `randn` functions. If extrinsic calls are enabled, the generated mex function uses the MATLAB global random number stream to generate random numbers. If extrinsic calls are not enabled, the MEX function built with `fiaccel` uses a self-contained random number generator.

If you disable extrinsic calls, the generated MEX function cannot display run-time messages from `error` or `assert` statements in your MATLAB code. The MEX function reports that it cannot display the error message. To see the error message, enable extrinsic function calls and generate the MEX function again.

`GenerateReport` — Create an HTML code generation report
`false` (default) | `true`

Whether to create an HTML code generation report, specified as `true` or `false`.

`GlobalDataSyncMethod` — MEX function global data synchronization with MATLAB global workspace
`SyncAlways` (default) | `SyncAtEntryAndExits` | `NoSync`

MEX function global data synchronization with MATLAB global workspace, specified as one of these values:

- `SyncAlways` — Synchronize global data at MEX function entry and exit and for all extrinsic calls to ensure maximum consistency between MATLAB and the generated MEX function. If the extrinsic calls do not affect global data, use this option in conjunction with the `coder.extrinsic -sync:off` option to turn off synchronization for these calls to maximize performance.
- `SyncAtEntryAndExits` — Global data is synchronized only at MEX function entry and exit. If your code contains extrinsic calls, but only a few affect global data, use this option in conjunction with the `coder.extrinsic -sync:on` option to turn on synchronization for these calls to maximize performance.
- `NoSync` — No synchronization occurs. Ensure that your MEX function does not interact with MATLAB globals before disabling synchronization otherwise inconsistencies between MATLAB and the MEX function might occur.

Data Types: `char`

`InlineStackLimit` — Stack size limit for inlined functions
`4000` (default) | positive integer

Stack size limit for inlined functions, specified as a positive integer.

`InlineThreshold` — Maximum size of functions to be inlined
`10` (default) | positive integer

Maximum size of functions to be inlined, specified as a positive integer.

`InlineThresholdMax` — Maximum size of functions after inlining
`200` (default) | positive integer

Maximum size of functions after inlining, specified as a positive integer.

IntegrityChecks — Memory integrity
true (default) | false

Set this property to true to detect any violations of memory integrity in code generated for MATLAB. When a violation is detected, execution stops and a diagnostic message displays. Set this property to false to disable both memory integrity checks and the runtime stack.

LaunchReport — Open the HTML code generation report automatically
true (default) | false

Whether to open the HTML code generation report automatically when code generation completes, specified as true or false. This property applies only if you set the GenerateReport property to true.

ReportPotentialDifferences — Report potential behavior differences between generated code and MATLAB code
true (default) | false

Whether to report potential behavior differences between generated code and MATLAB code, specified as true or false. If the ReportPotentialDifferences property is set to true, the code generation report has a tab that lists the potential differences. A potential difference is a difference that occurs at run time only under certain conditions.

ResponsivenessChecks — Enable responsiveness checks
true (default) | false

Whether to enable responsiveness checks, specified as true or false.

SaturateOnIntegerOverflow — Saturate on integer overflow
true (default) | false

Whether to saturate on integer overflow, specified as true or false. Overflows saturate to either the minimum or maximum value that the data type can represent. Set this property to true to have overflows saturate. Set this property to false to have overflows wrap to the appropriate value representable by the data type.

StackUsageMax — Maximum stack usage per application in bytes
200000 (default) | positive integer

Maximum stack usage per application in bytes, specified as a positive integer. Set a limit that is lower than the available stack size. Otherwise, a runtime stack overflow might occur. Overflows are detected and reported by the C compiler, not by fiaccel.

Examples

Create a Code Acceleration Configuration Object

This example shows how to use the coder.mexconfig function to create a coder.MexConfig configuration object. The object is set to disable run-time checks.

```
cfg = coder.mexconfig;
```

Turn off integrity checks, extrinsic calls, and responsiveness checks.

```
cfg.IntegrityChecks = false;
cfg.ExtrinsicCalls = false;
cfg.ResponsivenessChecks = false;
```

Use the `fiaccel` function to generate a MEX function for the file `foo.m`.

```
fiaccel -config cfg foo
```

Version History

Introduced in R2011a

R2023a: `ConstantInputs` property added to `coder.MexConfig` object

The `coder.mexconfig` object now includes the `ConstantInputs` property.

R2023a: `DynamicMemoryAllocation` Property To Be Removed

Behavior change in future release

In a future release, the `DynamicMemoryAllocation` property will be removed.

To dynamically allocate memory for variable-size arrays, use the `EnableDynamicMemoryAllocation` property. To configure the dynamic memory allocation threshold, use `DynamicMemoryAllocationThreshold` property.

See Also

`coder.ArrayType` | `coder.Constant` | `coder.EnumType` | `coder.FiType` | `coder.PrimitiveType` | `coder.StructType` | `coder.Type` | `coder.newtype` | `coder.resize` | `coder.typeof` | `fiaccel`

coder.SingleConfig class

Package: `coder`

Double-precision to single-precision conversion configuration object

Description

A `coder.SingleConfig` object contains the configuration parameters that the `convertToSingle` function requires to convert double-precision MATLAB code to single-precision MATLAB code. To pass this object to the `convertToSingle` function, use the `-config` option.

Construction

`scfg = coder.config('single')` creates a `coder.SingleConfig` object for double-precision to single-precision conversion.

Properties

`OutputFileNameSuffix` — Suffix for single-precision file name
`'_single'` (default) | character vector

Suffix that the single-conversion process uses for generated single-precision files.

`LogIOForComparisonPlotting` — Enable simulation data logging for comparison plotting of input and output variables
`false` (default) | true

Enable simulation data logging to plot the data differences introduced by single-precision conversion.

`PlotFunction` — Name of function for comparison plots
`''` (default) | character vector

Name of function to use for comparison plots.

To enable comparison plotting, set `LogIOForComparisonPlotting` to true. This option takes precedence over `PlotWithSimulationDataInspector`.

The plot function must accept three inputs:

- A structure that holds the name of the variable and the function that uses it.
- A cell array to hold the logged floating-point values for the variable.
- A cell array to hold the logged values for the variable after fixed-point conversion.

`PlotWithSimulationDataInspector` — Specify use of Simulation Data Inspector for comparison plots
`false` (default) | `true`

Use Simulation Data Inspector for comparison plots.

LogIOForComparisonPlotting must be set to true to enable comparison plotting. The PlotFunction option takes precedence over PlotWithSimulationDataInspector.

TestBenchName — Name of test file
'' (default) | character vector | cell array of character vectors

Test file name or names, specified as a character vector or cell array of character vectors. Specify at least one test file.

If you do not explicitly specify input parameter data types, the conversion uses the first file to infer these data types.

TestNumerics — Enable numerics testing
false (default) | true

Enable numerics testing to verify the generated single-precision code. The test file runs the single-precision code.

Methods

addFunctionReplacement	Replace double-precision function with single-precision function during single-precision conversion

Examples

Generate Single-Precision MATLAB Code

Create a coder.SingleConfig object.

```
scfg= coder.config('single');
```

Set the properties of the doubles-to-singles configuration object. Specify the test file. In this example, the name of the test file is myfunction_test. The conversion process uses the test file to infer input data types and collect simulation range data. Enable numerics testing and generation of comparison plots.

```
scfg.TestBenchName = 'myfunction_test';
scfg.TestNumerics = true;
scfg.LogIOForComparisonPlotting = true;
```

Run convertToSingle. Use the -config option to specify the coder.SingleConfig object that you want to use. In this example, the MATLAB function name is myfunction.

```
convertToSingle -config scfg myfunction
```

Version History

Introduced in R2015b

See Also

coder.config | convertToSingle

Topics
"Generate Single-Precision MATLAB Code"

DataTypeWorkflow.Converter

Create fixed-point converter object

Description

The `DataTypeWorkflow.Converter` object contains the object functions and parameters needed to collect simulation and derived data, propose and apply data types to the model, and analyze results. Use the `DataTypeWorkflow.Converter` object to perform the same fixed-point conversion tasks as the Fixed-Point Tool.

Creation

Syntax

```
converter = DataTypeWorkflow.Converter(systemToScale)
converter = DataTypeWorkflow.Converter(referencedModelSystem,'TopModel',
topModel)
```

Description

`converter = DataTypeWorkflow.Converter(systemToScale)` creates a converter object for the `systemToScale`.

`converter = DataTypeWorkflow.Converter(referencedModelSystem,'TopModel', topModel)` creates a converter object with the specified referenced model, `referencedModelSystem`, as the system to scale.

Input Arguments

`systemToScale` — Name of model or system to scale
character vector

Name of the model or subsystem to scale, specified as a character vector.

Example: `converter = DataTypeWorkflow.Converter('ex_fixed_point_workflow');`

`referencedModelSystem` — Name of referenced model or system inside a referenced model
character vector

Name of the referenced model or the subsystem within a referenced model to convert to fixed point, specified as a character vector.

`topModel` — Name of top-level model
character vector

Name of the top-level model that references `referencedModelSystem`, specified as a character vector. `topModel` is used during the range collection phase of conversion.

Properties

`CurrentRunName` — Current run in the converter object
character vector

Current run stored in the converter object, specified as a character vector.

Example: `converter.CurrentRunName = 'FixedPointRun'`

Data Types: `char`

`RunNames` — Names of all runs
cell array of character vectors

Names of all runs stored in the converter object, specified as a cell array of character vectors.

Example: `converter.RunNames`

Data Types: `cell`

`SelectedSystemToScale` — Name of model or subsystem
character vector

Name of the model or subsystem to scale, returned as a character vector.

Example: `converter.SelectedSystemToScale`

Data Types: `char`

`ShortcutsForSelectedSystem` — Available system shortcuts
cell array of character vectors

Available system settings shortcuts for the selected subsystem, specified as a cell array of character vectors.

Example: `converter.ShortcutsForSelectedSystem`

Data Types: `cell`

`TopModel` — Name of top-level model
character vector

Name of the top-level model that references `referencedModelSystem`, specified as a character vector. `topModel` is used during the range collection phase of conversion.

Example: `converter.TopModel`

Data Types: `char`

Object Functions

applyDataTypes	Apply proposed data types to model
applySettingsFromRun	Apply system settings used in previous run to model
applySettingsFromShortcut	Apply settings from shortcut to model
deriveMinMax	Derive range information for model
proposalIssues	Get results which have comments associated with them
proposeDataTypes	Propose data types for system
results	Find results for selected system in converter object

saturationOverflows	Get results where saturation occurred
simulateSystem	Simulate system specified by converter object
verify	Compare behavior of baseline and autoscaled systems
wrapOverflows	Get results where wrapping occurred

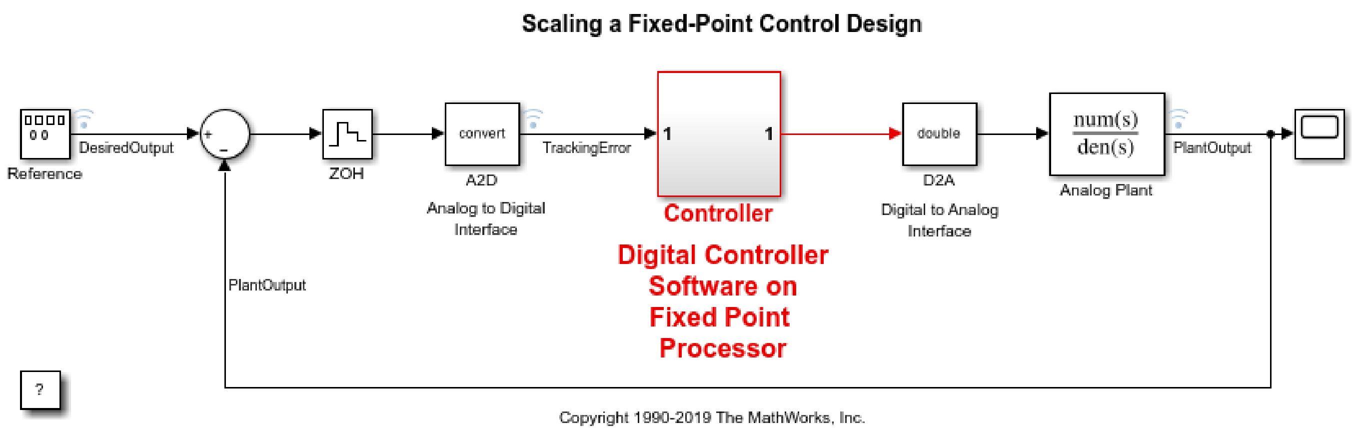
Examples

Create a `DataTypeWorkflow.Converter` Object

This example shows how to create a `DataTypeWorkflow.Converter` object.

Open the `fxpdemo_feedback` model.

```
open_system('fxpdemo_feedback');
```



The Controller subsystem uses fixed-point data types. Create a `DataTypeWorkflow.Converter` object.

```
converter = DataTypeWorkflow.Converter('fxpdemo_feedback/Controller');
```

You can view and edit properties of the `converter` object from the command line. For example, to change the name of the current run:

```
converter.CurrentRunName = 'FixedPointRun'
```

```
converter = 

  Converter with properties:

                CurrentRunName: 'FixedPointRun'
                      RunNames: {0x1 cell}
    ShortcutsForSelectedSystem: {6x1 cell}
                      TopModel: 'fxpdemo_feedback'
```

```
SelectedSystemToScale: 'fxpdemo_feedback/Controller'
```

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.ProposalSettings` | Fixed-Point Tool

Topics

“Convert a Model to Fixed Point Using the Command Line”
“The Command-Line Interface for the Fixed-Point Tool”

DataTypeWorkflow.ProposalSettings

Proposal settings object for data type proposals

Description

The `DataTypeWorkflow.ProposalSettings` object manages the properties related to how data types are proposed for a model, including the default floating point data type, and safety margins for the proposed data types.

Creation

Syntax

`propSettings = DataTypeWorkflow.ProposalSettings`

Description

`propSettings = DataTypeWorkflow.ProposalSettings` creates a proposal settings object.

Properties

`DefaultWordLength` — Default word length for floating-point signals
`16` (default) | scalar

Default word length for floating-point signals, specified as a scalar. Use this setting when the `ProposeFractionLength` property is set to `true`.

Example: `propSettings.DefaultWordLength = 16`

Data Types: `double`

`DefaultFractionLength` — Default fraction length for floating-point signals
`4` (default) | scalar

Default fraction length for floating-point signals, specified as a scalar. Use this setting when the `ProposeWordLength` property is set to `true`.

Example: `propSettings.DefaultFractionLength = 4`

Data Types: `double`

`ProposeFractionLength` — Whether to propose fraction lengths for specified word length
`true` (default) | `false`

Whether to propose fraction lengths for the default word length specified in the `DefaultWordLength` property, specified as a Boolean. Setting this property to `true` automatically sets the `ProposeWordLength` property to `false`.

Example: `propSettings.ProposeFractionLength = logical(true)`

Data Types: `logical`

`ProposeForInherited` — Whether to propose fixed-point data types for objects with an inherited output data type
`true` (default) | `false`

Whether to propose fixed-point data types for objects in the system with inherited output data types, specified as a Boolean.

Example: `propSettings.ProposeForInherited = logical(true)`

Data Types: `logical`

`ProposeForFloatingPoint` — Whether to propose fixed-point data types for objects with a floating-point output data type
`true` (default) | `false`

Whether to propose fixed-point data types for objects in the system with floating-point output data types, specified as a Boolean.

Example: `propSettings.ProposeForFloatingPoint = logical(true)`

Data Types: `logical`

`ProposeSignedness` — Whether to propose signedness for objects in the system
`true` (default) | `false`

Whether to propose signedness for objects in the system, specified as a Boolean.

The software bases the signedness proposal on collected range information and block constraints. Signals that are always strictly positive are assigned an unsigned data type proposal, and gain an additional bit of precision. If you set this property to `false`, the software proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the software will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

Example: `propSettings.ProposeForFloatingPoint = logical(true)`

Data Types: `logical`

`ProposeWordLength` — Whether to propose word lengths for specified default fraction lengths
`false` (default) | `true`

Whether to propose word lengths for the default fraction length in the `DefaultFractionLength` property, specified as a Boolean. Setting this property to `true` automatically sets the `ProposeFractionLength` property to `false`.

Example: `propSettings.ProposeWordLength = logical(false)`

Data Types: `logical`

`SafetyMargin` — Safety margin for simulation minimum and maximum values
`0` (default) | scalar

Safety margin for simulation minimum and maximum values, specified as a scalar.

The simulation minimum and maximum values are adjusted by the percentage designated by this parameter. This parameter allows you to specify a range different from that obtained from the simulation run.

For example, a value of `55` specifies that a range at least 55 percent larger is desired. A value of `-15` specifies that a range of up to 15 percent smaller is acceptable.

Example: `propSettings.SafetyMargin = 55`

Data Types: `double`

`UseDerivedMinMax` — Whether to use derived ranges to propose data types
`true` (default) | `false`

Whether to use derived ranges for data type proposals, specified as a Boolean.

Example: `propSettings.UseDerivedMinMax = logical(true)`

Data Types: `logical`

`UseSimMinMax` — Whether to use simulation ranges to propose data types
`true` (default) | `false`

Whether to use simulation ranges for data type proposals, specified as a Boolean.

Example: `propSettings.UseSimMinMax = logical(true)`

Data Types: `logical`

Object Functions

addTolerance	Specify numeric tolerance for converted system
clearTolerances	Clear all tolerances specified by a DataTypeWorkflow.ProposalSettings object
showTolerances	Show tolerances specified for a system

Alternatives

The properties of the `DataTypeWorkflow.ProposalSettings` object can also be controlled from the **Settings** menu in the Fixed-Point Tool. For more information, see Fixed-Point Tool.

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.Converter`

Topics

"Convert a Model to Fixed Point Using the Command Line"

DataTypeWorkflow.Result

Object containing run result information

Description

The `DataTypeWorkflow.Result` object manages the results of simulation, derivation, and data type proposals.

Creation

The `results` function returns a handle to a `DataTypeWorkflow.Result` object.

Properties

`Comments` — Comments associated with the signal
cell array of character vectors

Comments associated with the signal, specified as a cell array of character vectors.

Example: `results.Comments`

Data Types: `cell`

`CompiledDataType` — Data type used during simulation
character vector

Data type used during simulation, specified as a character vector.

Example: `results.CompiledDataType`

Data Types: `char`

`DerivedMax` — Derived maximum value
scalar

Derived maximum value for the signal or internal data based on specified design maximums, specified as a scalar.

Use the `DataTypeWorkflow.ProposalSettings` object and related object functions to specify and manage numeric tolerances for signals.

Example: `results.DerivedMax`

Data Types: `double`

`DerivedMin` — Derived minimum value
scalar

Derived minimum value for the signal or internal data based on specified design minimums, specified as a scalar.

Use the `DataTypeWorkflow.ProposalSettings` object and related object functions to specify and manage numeric tolerances for signals.

Example: `results.DerivedMin`

Data Types: `double`

`DesignMax` — Design maximum value
scalar

Design maximum value for the signal or internal data, specified as a scalar.

Example: `results.DesignMax`

Data Types: `double`

`DesignMin` — Design minimum value
scalar

Design minimum value for the signal or internal data, specified as a scalar.

Example: `results.DesignMin`

Data Types: `double`

`ProposedDataType` — Proposed data type
character vector

Proposed data type for the signal or internal data type associated with this result, specified as a character vector.

Example: `results.ProposedDataType`

Data Types: `char`

`ResultName` — Name of signal
character vector

Name of the signal or internal data associated with this result, specified as a character vector.

Example: `results.ResultName`

Data Types: `char`

`RunName` — Name of run associated with result
character vector

Name of the run associated with the result, specified as a character vector.

Example: `results.RunName`

Data Types: `char`

`Saturations` — Number of saturations that occurred
scalar

Number of saturations that occurred, specified as a scalar.

The number of occurrences where the signal or internal data associated with this result saturated at the maximum or minimum of its specified data type. The value of this property is the cumulative total of all of the run executions for this result.

Example: `results.Saturations`

Data Types: `double`

`SimMax` — Simulation maximum
scalar

Simulation maximum, specified as a scalar. This property represents the values obtained for the signal or internal data during all of the saved executions of the run this result is associated with.

Example: `results.SimMax`

Data Types: `double`

`SimMin` — Simulation minimum
scalar

Simulation minimum, specified as a scalar. This property represents the value obtained for the signal or internal data during all of the saved executions of the run this result is associated with.

Example: `results.SimMin`

Data Types: `double`

`SpecifiedDataType` — Specified data type of signal
character vector

Specified data type of the signal, specified as a character vector. This property takes effect the next time the system is run.

Example: `results.SpecifiedDataType`

Data Types: `char`

`Wraps` — Number of wraps that occurred
scalar

Number of wraps that occurred, specified as a scalar.

The number of occurrences where the signal or internal data associated with this result wrapped around the maximum or minimum of its specified data type. The value of this property is the cumulative total of all of the run executions for this result.

Example: `results.Wraps`

Data Types: `double`

Version History
Introduced in R2014b

See Also
`results` | `DataTypeWorkflow.Converter` | `DataTypeWorkflow.ProposalSettings`

Topics
"Convert a Model to Fixed Point Using the Command Line"

DataTypeWorkflow.VerificationResult

Verification results after converting a system to fixed point

Description

A `DataTypeWorkflow.VerificationResult` object contains the results after converting a system to fixed point. The verification result object indicates whether a conversion was successful based on the tolerances specified on the `DataTypeWorkflow.ProposalSettings` object used during the conversion.

Creation

Syntax

```
verificationResult = verify(converter,BaselineRunName,RunName)
```

Description

`verificationResult = verify(converter,BaselineRunName,RunName)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`, and stores the run information in a new run, `RunName`. It returns a `DataTypeWorkflow.VerificationResult` object that compares the baseline and verification runs.

The `DataTypeWorkflow.Converter` object contains instrumentation data from the run specified by `BaselineRunName`, as well as the tolerances specified on the associated `DataTypeWorkflow.ProposalSettings` object. The software determines if the behavior of the verification run is acceptable using the tolerances specified by the `ProposalSettings` object.

Properties

`RunName` — Name of verification run to create
character vector

Name of the verification run to create during the embedded simulation, specified as a character vector.

Example: `verificationResult.RunName`

Data Types: `char`

`BaselineRunName` — Baseline run to compare against
character vector

Baseline run to compare against, specified as a character vector.

Example: `verificationResult.BaselineRunName`

Data Types: `char`

`Status` — Whether the verification run meets the specified tolerances
`Pass` | `Warn` | `Fail`

Whether the verification run meets the specified tolerances, returned as either `Pass`, `Warn`, or `Fail`. For additional details, use `explore` to display logged data in the Simulation Data Inspector.

Status	Description
`Pass`	All signals with a specified tolerance on the associated `ProposalSettings` object are within the specified tolerances in the verification run.
`Fail`	One or more signals with a specified tolerance on the the associated `ProposalSettings` object are not within the specified tolerances in the verification run.

Example: `verificationResult.Status`

Data Types: `char`

Object Functions

explore   Explore comparison of baseline and fixed-point implementations

Version History

Introduced in R2019a

See Also

`DataTypeWorkflow.Converter` | `DataTypeWorkflow.ProposalSettings`

Topics

"Convert a Model to Fixed Point Using the Command Line"

fixed.DataGenerator

Creates value set and generates data

Description

Use the `fixed.DataSpecification` and `fixed.DataGenerator` objects to generate simulation inputs to test the full operating range of your designs.

Creation

Syntax

```
data = fixed.DataGenerator(Name, Value)
```

Description

`data = fixed.DataGenerator(Name, Value)` creates a `DataGenerator` object with additional properties specified as `Name, Value` pair arguments.

Properties

`DataSpecifications` — Properties of generated data
`fixed.DataSpecification` object | cell array of `fixed.DataSpecification` objects

Properties of the data to generate, specified as a `fixed.DataSpecification` object.

Specifying a cell array of `DataSpecification` objects produces a single `DataGenerator` object for input to a system with the same number of inputs and in the same order as elements in the cell array.

`NumDataPointsLimit` — Maximum number of data points in generated data
100000 (default) | integer-valued scalar

Maximum number of data points in generated data, specified as an integer-valued scalar. For more information, see `getNumDataPointsInfo`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Object Functions

getUniqueValues	Get unique values from fixed.DataGenerator object
getNumDataPointsInfo	Get information about number of data points in generated data
outputAllData	Get data from fixed.DataGenerator object

Examples

Create a `fixed.DataGenerator` object

Create a `DataGenerator` object by specifying a `DataSpecification` object in the constructor.

Create the `DataSpecification` object with an interval from -2π to 2π with a data type of `single`.

```
dataspec = fixed.DataSpecification('single',...
    'Intervals',{-2*pi, 2*pi})
```

```
dataspec = 
  fixed.DataSpecification with properties:

          DataTypeStr: 'single'
            Intervals: [-6.2832,6.2832]
     ExcludeDenormals: false
  ExcludeNegativeZero: false
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

Use the `DataSpecification` object to create a `DataGenerator` object. Limit the number of data points in the generated data to 5000 points. You can specify these properties as name-value pairs in the constructor of the `DataGenerator` object.

```
datagen = fixed.DataGenerator('DataSpecifications',dataspec,...
    'NumDataPointsLimit',5000)
```

```
datagen = 
  fixed.DataGenerator with properties:

    DataSpecifications: {[1x1 fixed.DataSpecification]}
    NumDataPointsLimit: 5000
```

Use the `outputAllData` function to see the generated data.

```
myData = outputAllData(datagen);
```

Algorithms

Data Generation for One-Dimensional, Two-Dimensional, and Complex Data

When you use a `DataGenerator` object to generate data for a `DataSpecification` object with the `Dimensions` property set to 1, the output data always contains the minimum and maximum values of the specified intervals, and any values specified by the `MandatoryValues` property.

When you generate data for a `DataSpecification` object with the `Dimensions` property set to a value greater than 1, the output is generated by taking a cartesian product of the one-dimensional output.

For example, consider the following two `DataSpecification` objects. The two objects are identical except that one is one-dimensional, and the other is two-dimensional.

```
dataspec_1d = fixed.DataSpecification('single',...
   'Intervals', {-1,1}, 'Dimensions',1);
```

```
dataspec_2d = fixed.DataSpecification('single',...
   'Intervals', {-1,1}, 'Dimensions',2);
```

Create two DataGenerator objects based on these specifications. Set the maximum number of data points in the generated data to inf.

```
datagen_1d = fixed.DataGenerator('DataSpecifications', ...
  dataspec_1d, 'NumDataPointsLimit', inf);
datagen_2d = fixed.DataGenerator('DataSpecifications', ...
  dataspec_2d, 'NumDataPointsLimit', inf);
```

Get the size of the generated data for each of the configurations.

```
size_1d_data = size(outputAllData(datagen_1d))
size_2d_data = size(outputAllData(datagen_2d))
```

```
size_1d_data =

     1   244

size_2d_data =

           2       59536
```

The length of the two-dimensional data is exactly the squared length of the one-dimensional data.

The DataGenerator generates complex data in a similar way to the two-dimensional data. Create a DataSpecification object with Dimensions set to 1 and the Complexity set to complex. Create a DataGenerator object using this specification.

```
dataspec_complex = fixed.DataSpecification('single', ...
'Intervals', {-1,1}, 'Dimensions', 1, 'Complexity', 'complex');


datagen_complex = fixed.DataGenerator('DataSpecifications', ...
  dataspec_complex, 'NumDataPointsLimit', inf);
```

Get the size of the generated data from this configuration.

```
size_complex_data = size(outputAllData(datagen_complex))
```

```
size_complex_data =

           1       59536
```

The length of the output data for the one-dimensional complex data is the same as the length of the two-dimensional real data.

Version History

Introduced in R2019b

See Also

Objects

fixed.DataSpecification | fixed.Interval

fixed.DataSpecification

Specify properties of data to generate

Description

Use the `fixed.DataSpecification` and `fixed.DataGenerator` objects to generate simulation inputs to test the full operating range of your designs.

Creation

Syntax

```
dataspec = fixed.DataSpecification(numerictype)
dataspec = fixed.DataSpecification(numerictype,Name,Value)
```

Description

`dataspec = fixed.DataSpecification(numerictype)` creates a `DataSpecification` object with default property values and data type specified by `numerictype`.

`dataspec = fixed.DataSpecification(numerictype,Name,Value)`creates a `DataSpecification` object with data type specified by `numerictype`, and additional properties specified as `Name,Value` pair arguments.

Input Arguments

`numerictype` — Data type of generated data
character vector | `Simulink.NumericType` object | `embedded.numerictype` object

Data type of the generated data, specified as a string or character vector that evaluates to a numeric data type, or as a `Simulink.NumericType` or `numerictype` object.

Example: `dataspec = fixed.DataSpecification('double')`

Example: `dataspec = fixed.DataSpecification('fixdt(1,16,4)')`

Example: `dataspec = fixed.DataSpecification(Simulink.NumericType);`

Properties

`DataTypeStr` — Data type of generated data
character vector | `Simulink.NumericType` object | `embedded.numerictype` object

Data type of the generated data, specified as a string or character vector that evaluates to a numeric data type, or as a `Simulink.NumericType` or `numerictype` object.

This property cannot be edited after construction.

`Intervals` — Intervals within which to generate numeric data
`fixed.Interval` object | array of `fixed.Interval` objects | cell array containing inputs to `fixed.Interval` constructor

Numeric intervals in which to generate numeric data, specified as a `fixed.Interval` object, an array of `fixed.Interval` objects, or a cell array containing inputs to the `fixed.Interval` constructor.

If you do not specify an interval, the default interval uses end points equal to the minimum and maximum representable values of the specified numeric type.

Example: `dataspec.Intervals = {-1,1};`

Example: `dataspec.Intervals = fixed.Interval(-1,1);`

`ExcludeDenormals` — Whether to exclude denormal numbers from generated data
false (default) | true

Whether to exclude denormal numbers from generated data, specified as a logical.

This property is only applicable when the `DataTypeStr` property is a floating-point type.

Data Types: `logical`

`ExcludeNegativeZero` — Whether to exclude negative zero from generated data
false (default) | true

Whether to exclude negative zero from generated data, specified as a logical.

This property is only applicable when the `DataTypeStr` property is a floating-point type.

Data Types: `logical`

`MandatoryValues` — Values to include in the generated data
`<empty>` (default) | scalar | vector | matrix | multidimensional array

Values to include in the generated data, specified as a scalar, vector, matrix, or multidimensional array. If the values specified in `MandatoryValues` are outside the range of the data type specified in `DataTypeStr`, the values are saturated to the nearest representable value.

Example: `dataspec.MandatoryValues = [-215, 216];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`Complexity` — Complexity of generated data
`'real'` (default) | `'complex'`

Complexity of the generated data, specified as either `'real'` or `'complex'`.

Example: `dataspec.Complexity = 'complex';`

Data Types: `char` | `string`

`Dimensions` — Dimension of the generated data
1 (default) | positive scalar integer | row vector of positive integers

Dimension of the generated data, specified as a positive scalar integer or row vector of positive integers.

Example: `dataspec.Dimensions = 3;`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Object Functions

contains	Determine whether value domain of a DataSpecification object contains a specified value
applyOnRootInport	(To be removed) Apply properties to Inport block

Examples

Create a `fixed.DataSpecification` object

Create a `fixed.DataSpecification` object with default property values and an int16 data type.

```
dataspec = fixed.DataSpecification('int16')
```

```
dataspec = 
  fixed.DataSpecification with properties:

          DataTypeStr: 'int16'
            Intervals: [-32768,32767]
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

The default interval of the `DataSpecification` object is equal to the range of the data type specified in the constructor.

Create a `fixed.DataSpecification` object from a `fixed.Interval` object

Create a `fixed.Interval` object specifying a range of -*π* to *π*.

```
interval = fixed.Interval(-pi,pi)
```

```
interval = 
    [-3.1416,3.1416]

  1x1 fixed.Interval with properties:

              LeftEnd: -3.1416
             RightEnd: 3.1416
         IsLeftClosed: true
        IsRightClosed: true
```

Create a `DataSpecification` object using this interval and a data type of `fixdt(1,16,10)`.

```
dataspec = fixed.DataSpecification('fixdt(1,16,10)',...
    'Intervals',interval)
```

```
dataspec = 
  fixed.DataSpecification with properties:
```

```
          DataTypeStr: 'sfix16_En10'
            Intervals: [-3.1416,3.1416]
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

Alternatively, you can specify the interval as a cell array of inputs to the `fixed.Interval` constructor. The following code generates an equivalent `DataSpecification` object.

```
dataspec = fixed.DataSpecification('fixdt(1,16,10)',...
    'Intervals',{-pi,pi})
```

```
dataspec = 
  fixed.DataSpecification with properties:

          DataTypeStr: 'sfix16_En10'
            Intervals: [-3.1416,3.1416]
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

Create a `DataSpecification` object that includes NaN and Inf

You can include NaN and Inf values in the generated data by specifying these values as intervals in an `Interval` object.

The following code creates a `DataSpecification` object that references an array of interval objects that include the values -Inf, Inf, NaN, and the range [-1, 1].

```
dataspec = fixed.DataSpecification('single', 'Intervals',...
 {{-Inf}, {Inf}, {NaN}, {-1,1}})
```

```
dataspec = 

  fixed.DataSpecification with properties:

          DataTypeStr: 'single'
            Intervals: [-Inf] [-1,1] [Inf] [NaN]
     ExcludeDenormals: false
  ExcludeNegativeZero: false
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

Version History

Introduced in R2019b

See Also

Objects
`fixed.DataGenerator` | `fixed.Interval`

fixed.Interval

Define interval of values

Description

A `fixed.Interval` object defines an interval of real-world values. Use the `Interval` object to specify a range of values in a `fixed.DataSpecification` object.

Creation

Syntax

```
interval = fixed.Interval
interval = fixed.Interval(a)
interval = fixed.Interval(a, b)
interval = fixed.Interval(a, b, endnotes)
interval = fixed.Interval(a, b, Name, Value)
interval = fixed.Interval(numerictype)
interval = fixed.Interval({___}, ...,{___})
```

Description

`interval = fixed.Interval` creates a unit interval, [0,1].

`interval = fixed.Interval(a)` creates a degenerate interval, containing only the value `a`.

`interval = fixed.Interval(a, b)` creates a closed interval from `a` to `b`.

`interval = fixed.Interval(a, b, endnotes)` creates an interval from `a` to `b`, with the `endnotes` argument specifying whether the interval is open or closed.

`interval = fixed.Interval(a, b, Name, Value)` creates an interval from `a` to `b` with the `IsLeftClosed` and `IsRightClosed` properties specified as `Name, Value` pair arguments.

`interval = fixed.Interval(numerictype)` creates an interval or array of intervals with end points equal to the minimum and maximum representable values of the specified numeric type.

`interval = fixed.Interval({___}, ...,{___})` returns an array of `Interval` objects, where each cell array specifies the arguments for one or more of the objects.

Input Arguments

a — Left endpoint of interval
scalar | vector

Left endpoint of interval, specified as a scalar or vector.

b — Right endpoint of interval
scalar | vector

Right endpoint of interval, specified as a scalar or vector.

endnotes — Whether the interval is open or closed
'[]' (default) | '[)' | '(]' | '()'

Argument indicating whether the interval is closed, open, or half-open, specified as one of the following character vectors.

Endnotes	Description
'[]'	Generates a closed set, which includes both of its endpoints.
'[)'	Generates a half-open interval, in which the first endpoint is included, but the second is not included in the set.
'(]'	Generates a half-open interval, in which the first endpoint is not included, but the second is included in the set.
'()'	Generates an open set, in which neither endpoint is included in the set.

Example: `interval = fixed.Interval(1, 10, '()');`

numerictype — Numeric data type
`Simulink.Numerictype` object | `embedded.numerictype` object | character vector

Numeric data type whose range of representable values defines the `Interval` object, specified as a `Simulink.Numerictype` object, an `embedded.numerictype` object, or a character vector representing a numeric data type, for example, `'single'`.

When `numerictype` is `'double'`, `'single'`, or `'half'`, the output `Interval` object is an array of 4 `Interval` objects with intervals `[-Inf]`, `[Inf]`, `[NaN]`, and `[-realmax, realmax]`. For more information on representable values of a data type, see `realmax`.

Example: `interval = fixed.Interval('fixdt(1,16,8)');`

Properties

LeftEnd — Left endpoint of interval
0 (default) | scalar

Left endpoint of interval, specified as a scalar.

This property cannot be edited after object creation.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

RightEnd — Right endpoint of interval
1 (default) | scalar

Right endpoint of interval, specified as a scalar.

This property cannot be edited after object creation.

Data Types: `half` | `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `fi`

`IsLeftClosed` — Whether the left end of the interval is closed
true (default) | false

Whether the left end of the interval is closed, specified as a logical value.

This property cannot be edited after object creation.

Data Types: `logical`

`IsRightClosed` — Whether the right end of the interval is closed
true (default) | false

Whether the right end of the interval is closed, specified as a logical value.

This property cannot be edited after object creation.

Data Types: `logical`

Object Functions

contains	Determine if one fixed.Interval object contains another
intersect	Intersection of fixed.Interval objects
isDegenerate	Determine whether the left and right ends of a fixed.Interval object are degenerate
isLeftBounded	Determine whether a fixed.Interval object is left-bounded
isRightBounded	Determine whether the a fixed.Interval object is right-bounded
isnan	Determine whether a fixed.Interval object is NaN
overlaps	Determine if two fixed.Interval objects overlap
quantize	Quantize interval to range of numeric data type
setdiff	Set difference of fixed.Interval objects
union	Union of fixed.Interval objects
unique	Get set of unique values in fixed.Interval object

Examples

Create a `fixed.Interval` object with default values

Create an `Interval` object with default property values. When you do not specify endpoint values, the `Interval` object uses endpoints 0 and 1.

```
interval = fixed.Interval()
```

```
interval = 
    [0,1]

  1x1 fixed.Interval with properties:

             LeftEnd: 0
            RightEnd: 1
        IsLeftClosed: true
       IsRightClosed: true
```

Create a degenerate interval

Create a degenerate interval, containing only a single point.

```
interval = fixed.Interval(pi)
```

```
interval = 
    [3.1416]

  1x1 fixed.Interval with properties:

              LeftEnd: 3.1416
             RightEnd: 3.1416
         IsLeftClosed: true
        IsRightClosed: true
```

This is equivalent to creating an interval with two equivalent endpoints.

```
interval = fixed.Interval(pi, pi)
```

```
interval = 
    [3.1416]

  1x1 fixed.Interval with properties:

              LeftEnd: 3.1416
             RightEnd: 3.1416
         IsLeftClosed: true
        IsRightClosed: true
```

Create an open interval

Specify end notes for an interval to create an open interval.

```
interval = fixed.Interval(-1, 1,'()') %#ok<*NASGU>
```

```
interval = 
    (-1,1)

  1x1 fixed.Interval with properties:

              LeftEnd: -1
             RightEnd: 1
         IsLeftClosed: false
        IsRightClosed: false
```

To create an interval that includes the first endpoint, but not the second, specify the end notes as `'[)'`

```
interval = fixed.Interval(-1, 1,'[)')
```

```
interval = 
    [-1,1)

  1x1 fixed.Interval with properties:
```

```
              LeftEnd: -1
             RightEnd: 1
         IsLeftClosed: true
        IsRightClosed: false
```

Create an interval with the range of a numeric data type

When you specify a numeric data type in the constructor of the `fixed.Interval` object, the range of the interval is set to the range of the data type.

Create an interval with the range of an `int8` data type.

```
interval_int8 = fixed.Interval('int8')
```

```
interval_int8 = 
    [-128,127]

  1x1 fixed.Interval with properties:

              LeftEnd: -128
             RightEnd: 127
         IsLeftClosed: true
        IsRightClosed: true
```

You can also specify a `Simulink.NumericType` to create an interval with the same range as the range representable by the `NumericType` object.

```
myNumericType = Simulink.NumericType;
myNumericType.DataTypeMode = "Fixed-point: binary point scaling";
myNumericType.Signedness = 'Unsigned';
myNumericType.WordLength = 16;
myNumericType.FractionLength = 14
```

```
myNumericType = 
  NumericType with properties:

      DataTypeMode: 'Fixed-point: binary point scaling'
        Signedness: 'Unsigned'
        WordLength: 16
    FractionLength: 14
           IsAlias: 0
         DataScope: 'Auto'
        HeaderFile: ''
       Description: ''
```

```
interval_16_14 = fixed.Interval(myNumericType)
```

```
interval_16_14 = 
    [0,3.9999]

  1x1 fixed.Interval with properties:

              LeftEnd: 0
             RightEnd: 3.9999
```

```
        IsLeftClosed: true
       IsRightClosed: true
```

Create an array of `fixed.Interval` objects

To create an array of `fixed.Interval` objects, in the constructor of the `Interval` object, you can specify a series of cell arrays, each of which contain the arguments of an `Interval` object.

```
intervalarray = fixed.Interval({-1,1},{5,10,'[)'},...
    {1000,1500,'IsLeftClosed',1,'IsRightClosed',0},...
    {'int8'})
```

```
intervalarray = 
    [-1,1]    [5,10)    [1000,1500)    [-128,127]

  1x4 fixed.Interval with properties:

             LeftEnd
            RightEnd
        IsLeftClosed
       IsRightClosed
```

Version History

Introduced in R2019b

See Also

Objects
fixed.DataGenerator | fixed.DataSpecification

LUTCompressionResult

Optimized lookup table data for all Lookup Table blocks in a system

Description

A `LUTCompressionResult` object contains the optimized lookup table data for all Lookup Table blocks in a system. To create a `LUTCompressionResult` object, use the `FunctionApproximation.compressLookupTables` function. To replace the lookup tables in your system with the optimized version, use the `replace` function.

Creation

Create a `LUTCompressionResult` object using `FunctionApproximation.compressLookupTables`.

Properties

`MemoryUnits` — Units for memory usage
'bytes' (default) | 'bits' | 'Kb' | 'Kibit' | 'KB' | 'KiB' | 'Mb' | 'Mibit' | 'MB' | 'MiB' | 'Gb' | 'Gibit' | 'GB' | 'GiB'

Units for `MaxMemoryUsage` property, specified as `'bits'`, `'bytes'`, or one of the other enumerated options.

Data Types: `char`

`MemoryUsageTable` — Table summarizing the effects of compression
table

Table summarizing the effects of compression. The table contains one row for each lookup table compressed in the system and its corresponding memory savings.

Data Types: `table`

`NumLUTsFound` — Number of lookup tables found in system
integer-valued scalar

Number of lookup tables found in the specified system, specified as an integer-valued scalar.

Data Types: `double`

`NumImprovements` — Number of lookup tables compressed
integer-valued scalar

Number of lookup tables compressed in the system, specified as an integer-valued scalar.

Data Types: `double`

`TotalMemoryUsed` — Total memory of all lookup tables in system before compression
scalar

Total memory of all lookup tables in the system before compression, returned as a scalar. You can specify the units of this property by using the `MemoryUnits` property.

Data Types: `double`

`TotalMemoryUsedNew` — Total memory of all lookup tables in system after compression
scalar

Total memory of all lookup tables in the system after compression, returned as a scalar. You can specify the units of this property by using the `MemoryUnits` property.

Data Types: `double`

`TotalMemorySavings` — Difference between total memory before compression and after compression
scalar

Difference between the total memory of all lookup tables in the system before and after compression, returned as a scalar. You can specify the units of this property by using the `MemoryUnits` property.

Data Types: `double`

`TotalMemorySavingsPercent` — Percentage reduction in memory used by lookup tables in the system
scalar

Percentage reduction in the memory used by the lookup tables in the system after compression, returned as a scalar.

Data Types: `double`

`SUD` — System containing compressed lookup tables
character vector

System containing compressed lookup tables, returned as a character vector. `SUD` is the same as the `system` input argument of the `FunctionApproximation.compressLookupTables` function.

Data Types: `char`

`WordLengths` — Word lengths used for breakpoints and table data in the compressed lookup tables
scalar | vector

Word lengths used for breakpoints and table data in the compressed lookup tables, returned as a scalar or vector of integers.

Data Types: `double`

`FindOptions` — Options for finding lookup tables in system
`Simulink.FindOptions` object

`Simulink.FindOptions` object specifying options for finding lookup tables in the system.

Object Functions

replace	Replace all Lookup Table blocks with compressed lookup tables
revert	Revert compressed Lookup Table blocks to original versions

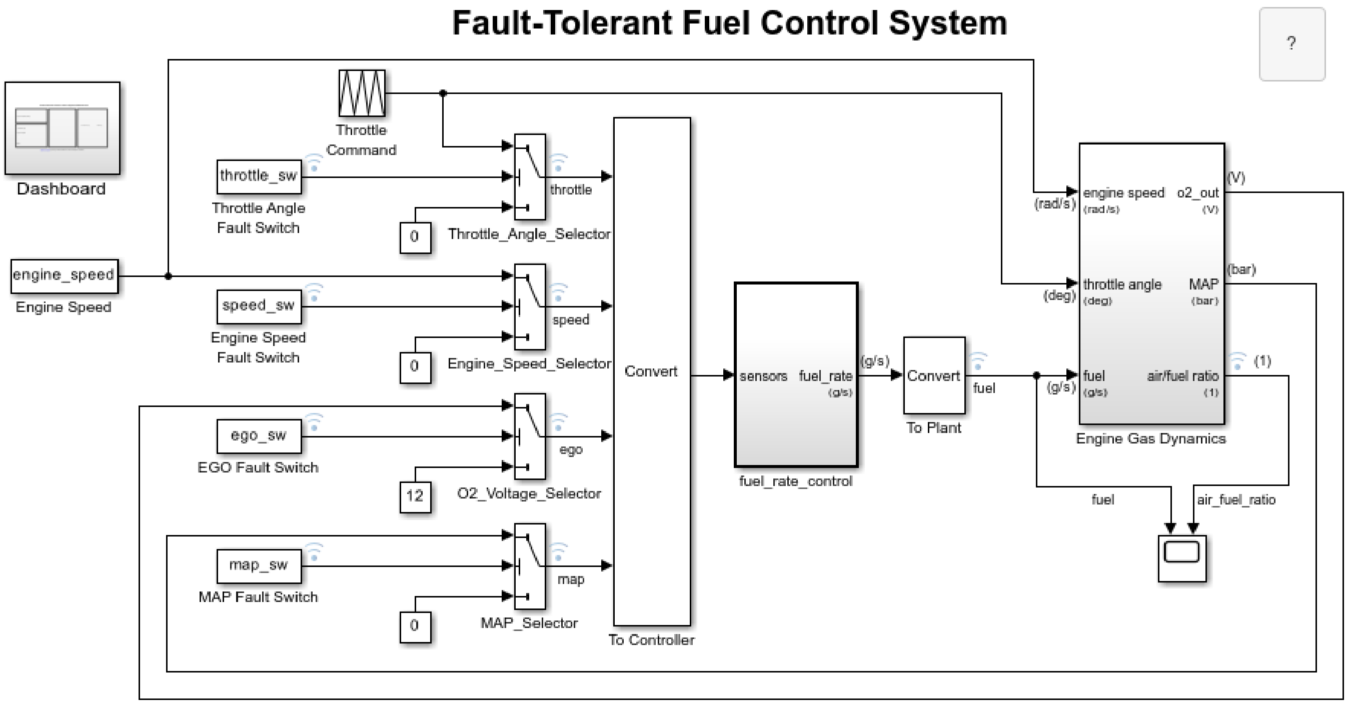
Examples

Compress All Lookup Table Blocks in a System

This example shows how to compress all Lookup Table blocks in a system.

Open the model containing the lookup tables that you want to compress.

```
system = 'sldemo_fuelsys';
open_system(system)
```



Open the Dashboard subsystem to simulate any combination of sensor failures.

Use the `FunctionApproximation.compressLookupTables` function to compress all of the lookup tables in the model. The output specifies all blocks that are modified and the memory savings for each.

```
compressionResult = FunctionApproximation.compressLookupTables(system)
```

```
- Found 5 supported lookup tables
- Percent reduction in memory for compressed solution
    -  2.37% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant
    -  2.37% for sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throt
    -  3.55% for sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estima
    -  6.38% for sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Es
    -  9.38% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki
```

```
compressionResult = 

  LUTCompressionResult with properties:

                 MemoryUnits: bytes
            MemoryUsageTable: [5x5 table]
                NumLUTsFound: 5
             NumImprovements: 5
             TotalMemoryUsed: 6024
          TotalMemoryUsedNew: 5796
          TotalMemorySavings: 228
   TotalMemorySavingsPercent: 3.7849
                         SUD: 'sldemo_fuelsys'
                 WordLengths: [8 16 32]
                 FindOptions: [1x1 Simulink.internal.FindOptions]
                     Display: 1
```

Use the `replace` function to replace each Lookup Table block with a block containing the original and compressed version of the lookup table.

```
replace(compressionResult);
```

You can revert the lookup tables back to their original state using the `revert` function.

```
revert(compressionResult);
```

Version History
Introduced in R2020a

See Also

Functions
`FunctionApproximation.compressLookupTables` | `replace` | `revert`

FunctionApproximation.ClassregProblem class

Package: FunctionApproximation

Specifies problem for the transform function

Description

The FunctionApproximation.ClassregProblem class creates a FunctionApproximation.CompactModelProblem object that specifies a problem for the transform function FunctionApproximation.TransformFunction. The properties of the FunctionApproximation.ClassregProblem class are read-only.

The FunctionApproximation.ClassregProblem class is a handle class.

Creation

Use the FunctionApproximation.TransformFunction class to create a TransformFunction object with the problem specified by FunctionApproximation.ClassregProblem.

Properties

Note Properties of FunctionApproximation.ClassregProblem are read-only.

FunctionToApproximate — Function to approximate
score transform function

Function to approximate with a lookup table approximation, specified as a score transform function. Use the FunctionApproximation.TransformFunction function to specify the trained classifier model object.

NumberOfInputs — Number of inputs to function approximation
1 | 2 | 3

Number of inputs to function approximation, specified as 1, 2, or 3. This property is inferred from the FunctionToApproximate property.

Data Types: double

InputTypes — Desired data types of inputs to function approximation
numerictype object

Desired data types of inputs to function approximation, specified as a numerictype object. Use the generateLearnerDataTypeFcn function to generate a function that defines the fixed-point data types.

InputLowerBounds — Lower limit of range of inputs to function to approximate
scalar | vector

Lower limit of range of inputs to function to approximate, specified as a scalar or vector. The lower bound of the input is determined by the input values.

Example: `InputLowerBounds = min(inputValues)`

`InputUpperBounds` — Upper limit of range of inputs to function to approximate
scalar | vector

Upper limit of range of inputs to function to approximate, specified as a scalar or vector. The upper bound of the input is determined by the input values.

Example: `InputUpperBounds = max(inputValues)`

`OutputType` — Desired data type of the function approximation output
`numerictype` object

Desired data type of the function approximation output, specified as a `numerictype` object. Use the `generateLearnerDataTypeFcn` function to generate a function that defines the fixed-point data types.

`Options` — Additional options and constraints to use in approximation
`FunctionApproximation.Options` object

Additional options and constraints to use in approximation, specified as a `FunctionApproximation.Options` object.

Version History

Introduced in R2023a

See Also

`FunctionApproximation.TransformFunction`

Topics

"Use Lookup Table to Approximate Score Transformation" (Statistics and Machine Learning Toolbox)

FunctionApproximation.LUTMemoryUsageCalculator class

Package: `FunctionApproximation`

Calculate memory used by lookup table blocks in a system

Description

The `FunctionApproximation.LUTMemoryUsageCalculator` class helps to calculate the memory used by each lookup table block, including 1-D Lookup Table, 2-D Lookup Table, and n-D Lookup Table, used in a model.

Construction

`calculator = FunctionApproximation.LUTMemoryUsageCalculator()` creates a `FunctionApproximation.LUTMemoryUsageCalculator` object. Use the `lutmemoryusage` method to calculate the memory used by each lookup table block in a model.

Properties

Public Properties

`FindOptions` — Options for finding lookup table blocks in a system
`Simulink.FindOptions` object

Options for finding lookup table blocks in a system, specified as a `Simulink.FindOptions` object.

Methods

lutmemoryusage	Calculate memory used by lookup table blocks in a system

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Calculate the Total Memory Used by Lookup Tables in a Model

Use the `FunctionApproximation.LUTMemoryUsageCalculator` class to calculate the total memory used by lookup table blocks in a model.

Create a `FunctionApproximation.LUTMemoryUsageCalculator` object.

```
calculator = FunctionApproximation.LUTMemoryUsageCalculator
```

Use the `lutmemoryusage` method to get the memory used by each lookup table block in the `sldemo_fuelsys` model.

```
openExample('simulink_automotive/ModelingAFaultTolerantFuelControlSystemExample',...
    'supportingfile','sldemo_fuelsys');
lutmemoryusage(calculator,'sldemo_fuelsys')

ans =

  5×2 table

                                             BlockPath
         _____________________________________________________________________________________

    1    "sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant"
    2    "sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throttle Esti
    3    "sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estimation"
    4    "sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Estimatio
    5    "sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki"
```

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
FunctionApproximation.Problem | FunctionApproximation.Options | FunctionApproximation.LUTSolution

Functions
solve | approximate | compare | totalmemoryusage | solutionfromID | displayfeasiblesolutions | displayallsolutions | lutmemoryusage

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

FunctionApproximation.LUTSolution class

Package: `FunctionApproximation`

Optimized lookup table data or lookup table data approximating a math function

Description

A `FunctionApproximation.LUTSolution` object contains optimized lookup table data or lookup table data approximating a math function. To create a `FunctionApproximation.LUTSolution` object, use the `solve` method on a `FunctionApproximation.Problem` object. To generate a subsystem containing the lookup table approximate or the optimized lookup table, or to generate the lookup table as a MATLAB function, use the `approximate` method of the `FunctionApproximation.LUTSolution` object.

You can save a `FunctionApproximation.LUTSolution` object to a MAT-file and restore the solution later.

Construction

`solution = solve(problem)` solves the problem defined by the `FunctionApproximation.Problem` object, `problem`, and returns the approximation or optimization, `solution`, as a `FunctionApproximation.LUTSolution` object.

Input Arguments

`problem` — Function to approximate, or lookup table to optimize
`FunctionApproximation.Problem` object

Function to approximate, or lookup table to optimize, and the constraints to consider during the optimization, specified as a `FunctionApproximation.Problem` object.

Properties

`ID` — ID of the solution
scalar integer

ID of the solution, specified as a scalar integer.

This property is read-only.

Data Types: `double`

`Feasible` — Whether the approximation meets the constraints
true | false

Whether the approximation or optimization specified by the `FunctionApproximation.LUTSolution` object, `solution`, meets the constraints specified in the `FunctionApproximation.Problem` object, `problem`, and its associated `FunctionApproximation.Options`.

This property is read-only.

Data Types: `logical`

`AllSolutions` — All solutions, including infeasible solutions
vector of `FunctionApproximation.LUTSolution` objects

All solutions found during the approximation, including infeasible solutions, specified as a vector of `FunctionApproximation.LUTSolution` objects.

This property is read-only.

`FeasibleSolutions` — All solutions that meet the constraints
vector of `FunctionApproximation.LUTSolution` objects

All solutions meeting the specified constraints, specified as a vector of `FunctionApproximation.LUTSolution` objects.

This property is read-only.

`PercentReduction` — Reduction in memory of lookup table
scalar

If the original `FunctionApproximation.Problem` object specified a lookup table block to optimize, the `PercentReduction` property indicates the reduction in memory from the original lookup table. If the original `FunctionApproximation.Problem` object specified a math function or function handle, the `PercentReduction` is `-Inf`.

This property is read-only.

Data Types: `double`

`SourceProblem` — Problem object approximated by the solution
`FunctionApproximation.Problem` object

`FunctionApproximation.Problem` object that the `FunctionApproximation.LUTSolution` object approximates.

This property is read-only.

`TableData` — Lookup table data
struct

Struct containing data related to lookup table approximation. The struct has the following fields.

- `BreakpointValues` - Breakpoints of the lookup table
- `BreakpointDataTypes`- Data type of the lookup table breakpoints
- `TableValues` - Values in the lookup table
- `TableDataType` - Data type of the table data
- `IsEvenSpacing` - Boolean value indicating if the breakpoints are evenly spaced.

This property is read-only.

Methods

approximate	Generate a Lookup Table block or lookup table as a MATLAB function from a `FunctionApproximation.LUTSolution`
compare	Compare numerical results of `FunctionApproximation.LUTSolution` to original function or lookup table
displayallsolutions	Display all solutions found during function approximation
displayfeasiblesolutions	Display all feasible solutions found during function approximation
getErrorValue	Get the total error of the lookup table approximation
replaceWithApproximate	Replace block with the generated lookup table approximation
revertToOriginal	Revert the block that was replaced by the approximation back to its original state
solutionfromID	Access a solution found during the approximation process
totalmemoryusage	Calculate total memory used by a lookup table approximation

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTMemoryUsageCalculator`

Functions
`solve` | `approximate` | `compare`

Topics
“Optimize Lookup Tables for Memory-Efficiency Programmatically”
“Optimize Lookup Tables for Memory-Efficiency”

FunctionApproximation.Options class

Package: FunctionApproximation

Specify additional options to use with FunctionApproximation.Problem object

Description

The FunctionApproximation.Options object contains additional options for defining a FunctionApproximation.Problem object.

Construction

options = FunctionApproximation.Options() creates a FunctionApproximation.Options object to use as an input to a FunctionApproximation.Problem object. The output, options, uses default property values.

options = FunctionApproximation.Options(Name,Value) creates a FunctionApproximation.Options object with property values specified by one or more Name,Value pair arguments. Name must appear inside single quotes (''). You can specify several name-value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Properties

AbsTol — Absolute tolerance of difference between original and approximate
non-negative scalar

Maximum tolerance of the absolute value of the difference between the original output value and the output value of the approximation, specified as a non-negative scalar.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fi

AllowUpdateDiagram — Whether to allow updating of the model diagram during the approximation process
true or 1 (default) | false or 0

Whether to allow updating of the model diagram during the approximation process, specified as a numeric or logical 1 (true) or 0 (false). This property is only relevant for FunctionApproximation.Problem objects that specify a Lookup Table block, or a Math Function block as the item to approximate.

Data Types: logical

ApproximateSolutionType — How to output optimized lookup table
'Simulink' (default) | 'MATLAB'

How to output optimized lookup table, specified as 'Simulink' or 'MATLAB'. When this property is set to 'Simulink', the approximate method produces a Simulink subsystem containing the optimized lookup table. When this property is set to 'MATLAB', the approximate method outputs the optimized lookup table as a MATLAB function.

Generating an optimized lookup table as a MATLAB function is not supported when:

- The `AUTOSARCompliant` property is set to `true`
- The `UseParallel` property is set to `true`
- The `HDLOptimized` property is set to `true`
- The `InterpolationMethod` property is set to `'None'`

Note The Simulink block and MATLAB function lookup table approximations generated by the `FunctionApproximation` package may not be exactly numerically equivalent. However, both solution forms are guaranteed to meet all constraints specified in the optimization problem.

Example: `options.ApproximateSolutionType ='MATLAB';`

Data Types: `char`

`AUTOSARCompliant` — Whether the generated lookup table block is an AUTOSAR block
`false` or `0` (default) | `true` or `1`

Whether the generated lookup table is AUTOSAR compliant, specified as a numeric or logical `1` (`true`) or `0` (`false`). When this property is set to `1` (`true`), the generated lookup table is a Curve or Map block from the AUTOSAR Blockset. When this property is set to `1` (`true`), the data type of the table data must equal the output data type of the block.

Setting this property to `1` (`true`) checks out a AUTOSAR Blockset license when you use the `approximate` or `replaceWithApproximate` methods.

This property is not supported when the `ApproximateSolutionType` property is set to `'MATLAB'`.

Data Types: `logical`

`BreakpointSpecification` — Spacing of breakpoint data
`ExplicitValues` (default) | `EvenSpacing` | `EvenPow2Spacing`

Spacing of breakpoint data, specified as one of the following values.

Breakpoint Specification	Description
`ExplicitValues`	Lookup table breakpoints are specified explicitly. Breakpoints can be closer together for some input ranges and farther apart in others.
`EvenSpacing`	Lookup table breakpoints are evenly spaced throughout.
`EvenPow2Spacing`	Lookup table breakpoints use power-of-two spacing. This breakpoint specification boasts the fastest execution speed because a bit shift can replace the position search.

For more information on how breakpoint specification can affect performance, see “Effects of Spacing on Speed, Error, and Memory Usage”.

Data Types: `char`

`Display` — Whether to display details of each iteration of the optimization
`true` or `1` (default) | `false` or `0`

Whether to display details of each iteration of the optimization, specified as a numeric or logical `1` (`true`) or `0` (`false`). A value of `1` (`true`) results in information in the command window at each iteration of the approximation process. A value of `0` (`false`) does not display information until the approximation is complete.

Data Types: `logical`

`ExploreHalf` — Whether to allow exploration of half precision
`true` or `1` (default) | `false` or `0`

Whether to allow the optimizer to explore half-precision data types for table data and breakpoints, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Data Types: `logical`

`HDLOptimized` — Whether to generate HDL-optimized approximate
`false` or `0` (default) | `true` or `1`

Whether to generate an HDL-optimized approximate, specified as a numeric or logical `1` (`true`) or `0` (`false`). A value of `1` (`true`) results in the approximate being a subsystem consisting of a prelookup step followed by interpolation that functions as a lookup table with explicit pipelining to generate efficient HDL code.

To generate an HDL-optimized approximate, the function to approximate must be one-dimensional and `BreakpointSpecification` must be set to `EvenSpacing` or `EvenPow2Spacing`.

This property is not supported when the `ApproximateSolutionType` property is set to `'MATLAB'`.

Data Types: `logical`

`Interpolation` — Method when an input falls between breakpoint values
`Linear` (default) | `Flat` | `Nearest` | `None`

When an input falls between breakpoint values, the lookup table interpolates the output value using neighboring breakpoints.

Interpolation Method	Description
`Linear`	Fits a line between the adjacent breakpoints, and returns the point on that line corresponding to the input.
`Flat`	Returns the output value corresponding to the breakpoint value that is immediately less than the input value. If no breakpoint value exists below the input value, it returns the breakpoint value nearest the input value.
`Nearest`	Returns the value corresponding to the breakpoint that is closest to the input. If the input is equidistant from two adjacent breakpoints, the breakpoint with the higher index is chosen.

Interpolation Method	Description
None	Generates a Direct Lookup Table (n-D) block, which performs table lookups without any interpolation or extrapolation. **Note** When generating a Direct Lookup Table block, the maximum number of inputs is two.

The interpolation method `None` is not supported when the `ApproximateSolutionType` property is set to `'MATLAB'`.

Data Types: `char`

`MaxMemoryUsage` — Maximum amount of memory the generated lookup table can use
80000000 (default) | scalar integer

The maximum amount of memory the generated lookup table can use, in bits, specified as a scalar integer. You can change the units of the option using the `MemoryUnits` property.

Data Types: `double`

`MaxTime` — Maximum amount of time for the approximation to run (in seconds)
`Inf` (default) | scalar

Maximum amount of time for the approximation to run, specified in seconds as a scalar number. The approximation runs until it reaches the time specified, finds an ideal solution, or reaches another stopping criteria.

Data Types: `double`

`MemoryUnits` — Units for maximum memory usage
`'bits'` (default) | `'bytes'` | `'Kb'` | `'Kibit'` | `'KB'` | `'KiB'` | `'Mb'` | `'Mibit'` | `'MB'` | `'MiB'` | `'Gb'` | `'Gibit'` | `'GB'` | `'GiB'`

Units for `MaxMemoryUsage` property, specified as `'bits'`, `'bytes'`, or one of the other enumerated options.

Data Types: `char`

`OnCurveTableValues` — Whether to constrain table values to the quantized output of the function being approximated
`false` or `0` (default) | `true` or `1`

Whether to constrain table values to the quantized output of the function being approximated, specified as a numeric or logical `1` (`true`) or `0` (`false`). By setting this property to `0` (`false`) and allowing off-curve table values, you may be able to reduce the memory of the lookup table while maintaining the same error tolerances, or maintain the same memory while reducing the error tolerances.

Data Types: `logical`

`RelTol` — Relative tolerance of difference between original and approximate
non-negative scalar

Maximum tolerance of the relative difference between the original output value and the output value of the approximation, specified as a non-negative scalar.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`SaturateToOutputType` — Saturate output of function to approximate to range of output type
`false` or `0` (default) | `true` or `1`

Whether to automatically saturate the range of the output of the function to approximate to the range of the output data type, specified as a numeric or logical `1` (`true`) or `0` (`false`).

Example: `options.SaturateToOutputType = 1;`

Data Types: `logical`

`UseParallel` — Whether to run iterations in parallel
`false` or `0` (default) | `true` or `1`

Whether to run iterations of the optimization in parallel, specified as a numeric or logical `1` (`true`) or `0` (`false`). Running the iterations in parallel requires a Parallel Computing Toolbox license. If you do not have a Parallel Computing Toolbox license, or if you specify `0` (`false`), the iterations run in serial.

This property is not supported when the `ApproximateSolutionType` property is set to `'MATLAB'`.

Example: `options.UseParallel = true;`

Data Types: `logical`

`WordLengths` — Word lengths permitted in the lookup table approximate
[8, 16, 32] (default) | integer scalar | integer vector

Specify the word lengths, in bits, that can be used in the lookup table approximate based on your intended hardware. For example, if you intend to target an embedded processor, you can restrict the data types in your lookup table to native types, 8, 16, and 32. The word lengths must be between 1 and 128.

Example: `options.WordLengths = [8,16,32];`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

Limitations

- Lookup table objects and breakpoint objects are not supported in a model mask workspace.

Algorithms

When you set `BreakpointSpecification` to `'ExplicitValues'`, during the approximation process, the algorithm also attempts to find a solution using `'EvenSpacing'` and `'EvenPow2Spacing'`. Likewise, when you set `BreakpointSpecification` to `'EvenSpacing'`, the algorithm also attempts to find a solution using `'EvenPow2Spacing'`. If you set the property to `'EvenPow2Spacing'`, the algorithm only attempts to find a solution using this spacing.

In cases where the `BreakpointSpecification` property is set to `'EvenSpacing'`, but the `InputUpperBounds` or `InputLowerBounds` property of the `FunctionApproximation.Problem` object is equal to the range of the `InputTypes`, the algorithm does not attempt to find a solution using `'EvenPow2Spacing'`.

Version History

Introduced in R2018a

R2021b: Generate an optimized lookup table approximation as a MATLAB function

You can now use the `FunctionApproximation.Problem` object to generate an optimized lookup table approximation as a MATLAB function. To generate MATLAB function, in a `FunctionApproximation.Options` object, set the `ApproximateSolutionType` property to `MATLAB`.

The generated MATLAB function is editable and supports C/C++ code generation using MATLAB Coder.

R2021a: Generate optimized one-dimensional lookup tables for HDL applications

Use lookup table optimization to generate a subsystem consisting of a prelookup step followed by interpolation that functions as a lookup table with explicit pipelining to generate efficient HDL code. To generate an HDL-optimized lookup table, set the HDLOptimized property to `true`.

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.LUTSolution` | `FunctionApproximation.LUTMemoryUsageCalculator`

Functions
`solve` | `approximate` | `compare` | `totalmemoryusage` | `solutionfromID` | `displayfeasiblesolutions` | `displayallsolutions` | `lutmemoryusage`

Topics
“Optimize Lookup Tables for Memory-Efficiency Programmatically”
“Optimize Lookup Tables for Memory-Efficiency”
“Generate an Optimized Lookup Table as a MATLAB Function Programmatically”
“Generate an Optimized Lookup Table as a MATLAB Function”

FunctionApproximation.Problem class

Package: `FunctionApproximation`

Object defining the function to approximate, or the lookup table to optimize

Description

The `FunctionApproximation.Problem` object defines the function to approximate with a lookup table, or the lookup table block to optimize. After defining the problem, use the `solve` method to generate a `FunctionApproximation.LUTSolution` object that contains the approximation.

Construction

`approximationProblem = FunctionApproximation.Problem()` creates a `FunctionApproximation.Problem` object with default property values. When no `function` input is provided, the `FunctionToApproximate` property is set to `'sin'`.

`approximationProblem = FunctionApproximation.Problem(function)` creates a `FunctionApproximation.Problem` object to approximate the function, Math Function block, or lookup table specified by `function`.

Input Arguments

`function` — Function or block to approximate, or lookup table block to optimize
`'sin'` (default) | math function | function handle | `cfit` object | Math Function block | Lookup Table block | Subsystem block

Function or block to approximate, or the lookup table block to optimize, specified as a function handle, a math function, a `cfit` object, a Simulink block or subsystem, or one of the lookup table blocks (for example, 1-D Lookup Table, n-D Lookup Table).

If you specify one of the lookup table blocks, the `solve` method generates an optimized lookup table.

If you specify a math function, a function handle, `cfit` object, or a block, the `solve` method generates a lookup table approximation of the input function.

If you specify a `cfit` object, use the `fittype` function to specify a library model to approximate. For a list of library models, see “List of Library Models for Curve and Surface Fitting” (Curve Fitting Toolbox).

Function handles must be on the MATLAB search path, or approximation fails.

The MATLAB math functions supported for approximation are:

- `1./x`
- `10.^x`
- `2.^x`
- `acos`

- acosh
- asin
- asinh
- atan
- atan2
- atanh
- cos
- cosh
- exp
- log
- log10
- log2
- sin
- sinh
- sqrt
- tan
- tanh
- x.^2

Tip The process of generating a lookup table approximation is faster for a function handle than for a subsystem. If a subsystem can be represented by a function handle, it is faster to approximate the function handle.

Data Types: char | function_handle

Properties

FunctionToApproximate — Function to approximate, or lookup table block to optimize
'sin' (default) | math function | function handle | cfit object | Math Function block | Lookup Table block | Subsystem block

Function or block to approximate, or the lookup table block to optimize, specified as a function handle, a math function, a Simulink block or subsystem, or one of the lookup table blocks (for example, 1-D Lookup Table, n-D Lookup Table).

If you specify one of the lookup table blocks, the solve method generates an optimized lookup table.

If you specify a cfit object, use the fittype function to specify a library model to approximate. For a list of library models, see “List of Library Models for Curve and Surface Fitting” (Curve Fitting Toolbox).

If you specify a math function, a function handle, cfit object, or a block, the solve method generates a lookup table approximation of the input function.

Function handles must be on the MATLAB search path, or approximation fails.

The MATLAB math functions supported for approximation are:

- `1./x`
- `10.^x`
- `2.^x`
- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`
- `atanh`
- `cos`
- `cosh`
- `exp`
- `log`
- `log10`
- `log2`
- `sin`
- `sinh`
- `sqrt`
- `tan`
- `tanh`
- `x.^2`

Tip The process of generating a lookup table approximation is faster for a function handle than for a subsystem. If a subsystem can be represented by a function handle, it is faster to approximate the function handle.

Data Types: `char` | `function_handle`

`NumberOfInputs` — Number of inputs to function approximation
1 | 2 | 3

Number of inputs to approximated function. This property is inferred from the `FunctionToApproximate` property, therefore it is not a writable property.

If you are generating a Direct Lookup Table, the function to approximate can have no more than two inputs.

Data Types: `double`

`InputTypes` — Desired data types of inputs to function approximation
`numerictype` object | vector of `numerictype` objects | `Simulink.Numerictype` object | vector of `Simulink.Numerictype` objects

Desired data types of the inputs to the approximated function, specified as a `numerictype`, `Simulink.Numerictype`, or a vector of `numerictype` or `Simulink.Numerictype` objects. The number of `InputTypes` specified must match the `NumberOfInputs`.

Example: `problem.InputTypes = ["numerictype(1,16,13)", "numerictype(1,16,10)"];`

`InputLowerBounds` — Lower limit of range of inputs to function to approximate
scalar | vector

Lower limit of range of inputs to function to approximate, specified as a scalar or vector. If you specify `inf`, the `InputLowerBounds` used during the approximation is derived from the `InputTypes` property. The dimensions of `InputLowerBounds` must match the `NumberOfInputs`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`InputUpperBounds` — Upper limit of range of inputs to function to approximate
scalar | vector

Upper limit of range of inputs to function to approximate, specified as a scalar or vector. If you specify `inf`, the `InputUpperBounds` used during the approximation is derived from the `InputTypes` property. The dimensions of `InputUpperBounds` must match the `NumberOfInputs`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`OutputType` — Desired data type of the function approximation output
`numerictype` | `Simulink.Numerictype`

Desired data type of the function approximation output, specified as a `numerictype` or `Simulink.Numerictype`. For example, to specify that you want the output to be a signed fixed-point data type with 16-bit word length and best-precision fraction length, set the `OutputType` property to `"numerictype(1,16)"`.

Example: `problem.OutputType = "numerictype(1,16)";`

`Options` — Additional options and constraints to use in approximation
`FunctionApproximation.Options` object

Additional options and constraints to use in approximation, specified as a `FunctionApproximation.Options` object.

Methods

solve	Solve for optimized solution to function approximation problem

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create Problem Object to Approximate a Function Handle

Create a `FunctionApproximation.Problem` object, specifying a function handle that you want to approximate.

```
problem = FunctionApproximation.Problem(@(x,y) sin(x)+cos(y))
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x,y)sin(x)+cos(y)
           NumberOfInputs: 2
               InputTypes: ["numerictype('double')"    "numerictype('double')"]
         InputLowerBounds: [-Inf -Inf]
         InputUpperBounds: [Inf Inf]
               OutputType: "numerictype('double')"
                  Options: [1×1 FunctionApproximation.Options]
```

The `FunctionApproximation.Problem` object, `problem`, uses default property values.

Set the range of the function inputs to be between zero and `2*pi`.

```
problem.InputLowerBounds = [0,0];
problem.InputUpperBounds = [2*pi, 2*pi]
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x,y)sin(x)+cos(y)
           NumberOfInputs: 2
               InputTypes: ["numerictype('double')"    "numerictype('double')"]
         InputLowerBounds: [0 0]
         InputUpperBounds: [6.2832 6.2832]
               OutputType: "numerictype('double')"
                  Options: [1×1 FunctionApproximation.Options]
```

Create Problem Object to Approximate a Math Function

Create a `FunctionApproximation.Problem` object, specifying a math function to approximate.

```
problem = FunctionApproximation.Problem('log')
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x)log(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(1,16,10)"
         InputLowerBounds: 0.6250
         InputUpperBounds: 15.6250
               OutputType: "numerictype(1,16,13)"
                  Options: [1×1 FunctionApproximation.Options]
```

The math functions have appropriate input range, input data type, and output data type property defaults.

Create Problem Object to Approximate a Curve Fitting Object

Create a `FunctionApproximation.Problem` object, specifying a `cfit` object to approximate.

```
ffun = fittype('exp1');
cfun = cfit(ffun,0.1,0.2);
problem = FunctionApproximation.Problem(cfun);
```

```
problem = 

  1×1 FunctionApproximation.Problem with properties:

    FunctionToApproximate: [1x1 cfit]
           NumberOfInputs: 1
               InputTypes: "numerictype('double')"
         InputLowerBounds: -Inf
         InputUpperBounds: Inf
               OutputType: "numerictype('double')"
                  Options: [1×1 FunctionApproximation.Options]
```

Create Problem Object to Optimize a Lookup Table Block

Create a `FunctionApproximation.Problem` object to optimize an existing lookup table.

```
openExample('simulink_automotive/ModelingAFaultTolerantFuelControlSystemExample',...
    'supportingfile','sldemo_fuelsys');
problem = FunctionApproximation.Problem('sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Co
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: 'sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant'
           NumberOfInputs: 2
               InputTypes: ["numerictype('single')"    "numerictype('single')"]
         InputLowerBounds: [50 0.0500]
         InputUpperBounds: [1000 0.9500]
               OutputType: "numerictype('single')"
                  Options: [1×1 FunctionApproximation.Options]
```

The software infers the properties of the `problem` object from the model.

Search for Floating-Point Solution to Function Approximation Problem

Since R2023a

This example shows how to search for pure floating-point solutions to the function approximation problem.

Create a `FunctionApproximation.Problem` object specifying a function to approximate.

```
problem = FunctionApproximation.Problem("sin");
```

Specify the input and output types to be a floating-point data type.

```
problem.InputTypes = [numerictype('Single')];
problem.OutputType = [numerictype('Single')];
```

Use the `FunctionApproximation.Options` object to specify wordlengths that can be used in the lookup table approximation. To search for floating-point solutions, specify word lengths corresponding to a single-precision or double-precision data type.

```
problem.Options.WordLengths = 32;
```

Use the `solve` method to generate an approximation of the function.

```
solve(problem)
```

```
Searching for fixed-point solutions.

|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   0 |            128 |        0 |          2 |              32 |           32 |             Eve
|   1 |           1568 |        1 |         47 |              32 |           32 |             Eve
|   2 |           1536 |        1 |         46 |              32 |           32 |             Eve
|   3 |           1216 |        1 |         36 |              32 |           32 |             Eve
|   4 |           1184 |        1 |         35 |              32 |           32 |             Eve
|   5 |            832 |        1 |         24 |              32 |           32 |             Eve
|   6 |            800 |        1 |         23 |              32 |           32 |             Eve
|   7 |            448 |        0 |         12 |              32 |           32 |             Eve
|   8 |            608 |        0 |         17 |              32 |           32 |             Eve
|   9 |            704 |        0 |         20 |              32 |           32 |             Eve
|  10 |            736 |        0 |         21 |              32 |           32 |             Eve
|  11 |            768 |        1 |         22 |              32 |           32 |             Eve
|  12 |            128 |        0 |          2 |              32 |           32 |        EvenPow
|  13 |           1152 |        1 |         18 |              32 |           32 |          Explic
|  14 |           1024 |        0 |         16 |              32 |           32 |          Explic
|  15 |           1152 |        0 |         18 |              32 |           32 |          Explic
|  16 |           1280 |        1 |         20 |              32 |           32 |          Explic
Searching for floating-point solutions.

|  17 |           1536 |        1 |         46 |              32 |           32 |             Eve
|  18 |            128 |        0 |          2 |              32 |           32 |        EvenPow
|  19 |           1152 |        1 |         18 |              32 |           32 |          Explic
|  20 |           1024 |        0 |         16 |              32 |           32 |          Explic

Best Solution
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  11 |            768 |        1 |         22 |              32 |           32 |             Eve

ans = 
  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 11
    Feasible: "true"
```

The `solve` method returns all feasible solutions. In the table, fixed-point solutions are returned first, followed by floating-point solutions. The Lookup Table Optimizer selects a floating-point solution as the best solution when all of these conditions are met:

- The floating-point solution requires equal or less memory than a fixed-point solution.

- Both the `InputTypes` and `OutputType` properties of the `FunctionApproximation.Problem` object specify a floating-point data type.
- The `WordLengths` property of the `FunctionApproximation.Options` object includes word lengths corresponding to a single-precision or double-precision data type.

Limitations

- Lookup table objects and breakpoint objects are not supported in a model mask workspace.

Algorithms

Required Specifications

Functions and function handles that you approximate must meet the following criteria.

- The function must be time-invariant.
- The function must operate element-wise, meaning for each input there is one output.
- The function must not contain states.

For more information, see “Vectorization”.

Infinite Upper and Lower Input Bounds

When a `Problem` object specifies infinite input ranges and the input type is non-floating-point, during the approximation, the software infers upper and lower ranges based on the range of the input data type. The resulting `FunctionApproximation.LUTSolution` object specifies the bounds that the algorithm used during the approximation, not the originally specified infinite bounds.

Upper and Lower Input Bounds and Input Data Type Range

If the `InputLowerBounds` or `InputUpperBounds` specified for a `Problem` object fall outside the range of the specified `InputTypes`, the algorithm uses the range of the data type specified by `InputTypes` for the approximation.

In cases where the `BreakpointSpecification` property of the `FunctionApproximation.Options` object is set to `'EvenSpacing'`, but the `InputUpperBounds` or `InputLowerBounds` property of the `FunctionApproximation.Problem` object is equal to the range of the `InputTypes`, the algorithm does not attempt to find a solution using `'EvenPow2Spacing'`.

Version History

Introduced in R2018a

R2022a: Support for curve fitting objects

The `FunctionApproximation.Problem` object now supports curve fitting `cfit` objects as valid inputs for approximation.

R2022a: Improved memory reduction for 1D and flat interpolation

The **Lookup Table Optimizer** has an improved algorithm for lookup table value and breakpoint optimization for one-dimensional functions with flat interpolation. This enhancement can enable improved memory reduction of the optimized lookup table and faster completion of the lookup table optimization process.

This improvement applies when the function to approximate is one-dimensional and all of these options are specified in `FunctionApproximation.Options`:

- `Interpolation` is set to `Flat`.
- `BreakpointSpecification` is set to `ExplicitValues`.
- `OnCurveTableValues` is set to `false`.

R2021b: Generate an optimized lookup table approximation as a MATLAB function

You can now use the `FunctionApproximation.Problem` object to generate an optimized lookup table approximation as a MATLAB function. To generate MATLAB function, in a `FunctionApproximation.Options` object, set the `ApproximateSolutionType` property to `MATLAB`.

The generated MATLAB function is editable and supports C/C++ code generation using MATLAB Coder.

R2021a: Lookup table optimization support for functions with scalar inputs

Previously, the `FunctionApproximation.Problem` class required that functions and function handles to approximate were vectorized, meaning that for each input, there is exactly one output. Lookup table optimization now fully supports approximation of Simulink blocks and subsystems that only allow scalar inputs.

R2021a: Improved lookup table value optimization

The **Lookup Table Optimizer** has an improved algorithm for lookup table value optimization for the `Flat` and `Nearest` interpolation methods when off-curve table values are allowed. This enhancement can enable faster completion of the lookup table optimization process and improved memory reduction of the optimized lookup table.

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Options` | `FunctionApproximation.LUTSolution` | `FunctionApproximation.LUTMemoryUsageCalculator`

Functions
`solve` | `approximate` | `compare`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"

“Optimize Lookup Tables for Memory-Efficiency”
“Generate an Optimized Lookup Table as a MATLAB Function Programmatically”
“Generate an Optimized Lookup Table as a MATLAB Function”
“Optimize Lookup Tables for Periodic Functions”

FunctionApproximation.TransformFunction class

Package: `FunctionApproximation`

Object defining transform function to approximate

Description

Use the `FunctionApproximation.TransformFunction` object to define a score transformation function of a trained classifier to approximate with an optimized lookup table to use for fixed-point code generation. After defining the `FunctionApproximation.TransformFunction` object, use the `approximate` method to solve for the lookup table approximation.

Creation

Description

`approxObj = FunctionApproximation.TransformFunction(filename,X,T)`

Input Arguments

`filename` — Name of MAT-file that contains structure array representing model object
character vector | string scalar

Name of MAT-file that contains structure array representing a model object, specified as a character vector or string scalar.

After training a machine learning model, use the `saveLearnerForCoder` function to create `filename`. The extension of `filename` must be `.mat`.

Example: `saveLearnerForCoder(model,'filename');`

Data Types: `char` | `string`

X — Predictor data
numeric matrix

Predictor data for the `predict` function of the model stored in `filename`, specified as an *n*-by-*p* numeric matrix, where *n* is the number of observations and *p* is the number of predictor variables.

The data set `X` used in `FunctionApproximation.TransformFunction` must be the same data that was passed to the `generateLearnerDataTypeFcn` function.

Data Types: `single` | `double`

T — Data types for fixed-point code generation
structure

Data types for fixed-point code generation, specified as a structure.

Use the `generateLearnerDataTypeFcn` function to generate the `myMdl_datatype` function. Then, use the `myMdl_datatype` function to create a structure that defines the fixed-point data types.

Example: `generateLearnerDataTypeFcn('myMdl',[X; XTest]); T = myMdl_datatype('Fixed');`

Properties

`Problem` — Score transformation function to approximate with lookup table
`FunctionApproximation.ClassregProblem` class

Score transformation function to approximate with a lookup table, specified as a `FunctionApproximation.ClassregProblem` class. The software defines the `Problem` property when you create a `FunctionApproximation.TransformFunction` object.

Tips

- If you use a nondefault score transformation function such as `doublelogit`, `logit`, or `symmetriclogit` for the trained classifier, you can use `FunctionApproximation.TransformFunction` and `approximate` to update the fixed-point data type structure to include a lookup table that approximates the score transformation function.

Version History

Introduced in R2023a

See Also

`approximate` | `FunctionApproximation.ClassregProblem`

Topics

"Use Lookup Table to Approximate Score Transformation" (Statistics and Machine Learning Toolbox)

fxpOptimizationOptions class

Specify options for data type optimization

Description

The `fxpOptimizationOptions` object enables you to specify options and constraints to use during the data type optimization process.

Construction

`opt = fxpOptimizationOptions()` creates a `fxpOptimizationOptions` object with default values.

`opt = fxpOptimizationOptions(Name,Value)` creates an `fxpOptimizationOptions` object with property values specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`MaxIterations` — Maximum number of iterations to perform
50 (default) | scalar integer

Maximum number of iterations to perform, specified as a scalar integer. The optimization process iterates through different solutions until it finds an ideal solution, reaches the maximum number of iterations, or reaches another stopping criteria.

Example: `opt.MaxIterations = 75;`

Data Types: `double`

`MaxTime` — Maximum amount of time for the optimization to run (in seconds)
600 (default) | scalar

Maximum amount of time for the optimization to run, specified in seconds as a scalar number. The optimization runs until it reaches the time specified, an ideal solution, or another stopping criteria.

Example: `opt.MaxTime = 1000;`

Data Types: `double`

`Patience` — Maximum number of iterations where no new best solution is found
10 (default) | scalar integer

Maximum number of iterations where no new best solution is found, specified as a scalar integer. The optimization continues as long as the algorithm continues to find new best solutions.

Example: `opt.Patience = 15;`

Data Types: `double`

Verbosity — Level of information displayed at the command line during the optimization
'High' (default) | 'Moderate' | 'Silent'

The level of information displayed at the command line during the optimization process, specified as either 'High', 'Moderate', or 'Silent'.

- 'Silent' - Nothing is displayed at the command line until the optimization process is finished
- 'Moderate' - Information is displayed at each major step of the optimization process, including when the process is in the preprocessing, modeling, and optimization phases.
- 'High' - Information is displayed at the command line at each iteration of the optimization process, including whether a new best solution was found, and the cost of the solution.

Example: `opt.Verbosity = 'Moderate';`

Data Types: char | string

AllowableWordLengths — Word lengths that can be used in your optimized system under design
[2:128] (default) | scalar integer | vector of integers

Specify the word lengths that can be used in your optimized system under design. Use this property to target the neighborhood search of the optimization process. The final result of the optimization uses word lengths in the intersection of the AllowableWordLengths and word lengths compatible with hardware constraints specified in the **Hardware Implementation** pane of your model.

Example: `opt.AllowableWordLengths = [8:11,16,32];`

Data Types: double

ObjectiveFunction — Objective function to use during optimization search
'BitWidthSum' (default) | 'OperatorCount'

Objective function to use during optimization search, specified as one of these values:

- 'BitWidthSum' — Minimize total bit width sum.
- 'OperatorCount' — Minimize estimated count of operators in generated C code.

  This option may result in a lower program memory size for C code generated from Simulink models. The 'OperatorCount' objective function is not suitable for FPGA or ASIC targets.

  Note To use 'OperatorCount' as the objective function during optimization, the model must be ready for code generation. For more information about determining code generation readiness, see “Check Model and Configuration for Code Generation” (Embedded Coder).

Data Types: char

UseParallel — Whether to run iterations in parallel
false (default) | true

Whether to run iterations of the optimization in parallel, specified as a logical. Running the iterations in parallel requires a Parallel Computing Toolbox license. If you do not have a Parallel Computing Toolbox license, or if you specify false, the iterations run in serial.

Data Types: logical

`AdvancedOptions` — Additional options for optimization
object

Additional advanced options for optimization. `AdvancedOptions` is an object containing additional properties that can affect the optimization.

Property	Description
`PerformNeighborhoodSearch`	• `1` (default) – Perform a neighborhood search for the optimized solution. • `0` – Do not perform a neighborhood search. Selecting this option can increase the speed of the optimization process, but also increases the chances of finding a less ideal solution.
`EnforceLooseCoupling`	Some blocks have a parameter that forces inputs to share a data type, or forces the output to share the same data type as the input. • `1` (default) – Allow the optimizer to relax this restriction on all blocks in the system under design. Relaxing this restriction enables the optimizer to provide better fitting data types. • `0` – Do not allow the optimizer to relax this restriction on blocks in the system under design.
`UseDerivedRangeAnalysis`	• `0` (default) – The optimizer does not consider ranges derived from design ranges in the model when assessing a solution. • `1` – The optimizer considers both observed simulation ranges and ranges derived from design ranges in the model when assessing a solution. Depending on the model configuration, derived range analysis may take longer than simulation of the model.
`SimulationScenarios`	Define additional simulation scenarios to consider during optimization using a `Simulink.SimulationInput` object. For an example, see “Optimize Data Types Using Multiple Simulation Scenarios”.
`SafetyMargin`	Enter a safety margin, specified as a positive scalar value indicating the percentage increase in the bounds of the collected range. The safety margin is applied to the union of all collected ranges, including simulation ranges, derived ranges, and design ranges.

Property	Description
`DataTypeOverride`	Override data types specified in the model when simulating during the range collection phase of optimization. • `'Off'` (default) – Do not override data types • `'Single'` – Override data types with singles • `'Double'` – Override data types with doubles • `'ScaledDouble'` – Override data types with scaled doubles
`HandleUnsupported`	Some blocks are not supported for fixed-point conversion. For more information, see “Blocks That Do Not Support Fixed-Point Data Types”. • `'Isolate'` (default) – Isolate unsupported blocks with Data Type Conversion blocks. Isolated blocks are ignored by the optimizer. • `'Error'` – Stop optimization and report an error when the system contains blocks that are not supported for fixed-point conversion. • `'Warn'` – Warn when the system contains blocks that are not supported for fixed-point conversion. Ignore unsupported blocks and continue optimization. This option allows you to replace unsupported constructs with other solutions, such as lookup tables, after optimization is complete.
`PerformSlopeBiasCancellation`	• `0` (default) – Do not propagate slope-bias data types. • `1` – Propagate slope-bias data types from outside the system under design. Slopes and biases are chosen to reduce the complexity of generated code.
`InstrumentationContext`	`[model '/Subsystem']` – Restrict instrumentation for minimum, maximum, and overflow logging for the range collection step of optimization to a subsystem. The subsystem must be under the top-level model and contain the system under design.

Methods

addSpecification	Specify known data types in a system
addTolerance	Specify numeric tolerance for optimized system
showSpecifications	Show specifications for a system
showTolerances	Show tolerances specified for a system

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Create an fxpOptimizationOptions Object

Create an `fxpOptimizationObject` with default property values.

```
options = fxpOptimizationOptions();
```

Edit the properties after creation using dot syntax.

```
options.Patience = 15;
options.AllowableWordLengths = [8,16,32];
options.AdvancedOptions.UseDerivedRangeAnalysis = true
```

```
options = 
  fxpOptimizationOptions with properties:

           MaxIterations: 50
                 MaxTime: 600
                Patience: 15
               Verbosity: High
    AllowableWordLengths: [8 16 32]
       ObjectiveFunction: BitWidthSum
             UseParallel: 0

   Advanced Options
         AdvancedOptions: [1x1 DataTypeOptimization.AdvancedFxpOptimizationOptions]
```

Create an fxpOptimizationOptions Object With Non-Default Settings

Use property name-value pairs to set properties at object creation.

```
options = fxpOptimizationOptions('Patience',15,'AllowableWordLengths',[8,16,32])
```

```
options = 
  fxpOptimizationOptions with properties:

           MaxIterations: 50
                 MaxTime: 600
                Patience: 15
               Verbosity: High
    AllowableWordLengths: [8 16 32]
       ObjectiveFunction: BitWidthSum
             UseParallel: 0

   Advanced Options
         AdvancedOptions: [1x1 DataTypeOptimization.AdvancedFxpOptimizationOptions]
```

Specify advanced options.

```
options.AdvancedOptions.UseDerivedRangeAnalysis = 1
```

```
options = 
  fxpOptimizationOptions with properties:

           MaxIterations: 50
                 MaxTime: 600
                Patience: 15
               Verbosity: High
    AllowableWordLengths: [8 16 32]
       ObjectiveFunction: BitWidthSum
             UseParallel: 0

   Advanced Options
         AdvancedOptions: [1x1 DataTypeOptimization.AdvancedFxpOptimizationOptions]
```

Import an fxpOptimizationOptions Object into Fixed-Point Tool

You can import an `fxpOptimizationOptions` object into the Fixed-Point Tool to perform data type optimization in the app. By importing an `fxpOptimizationOptions` object rather than specifying settings manually in the app, you can easily save and restore your settings.

Open the model.

```
model = 'ex_controllerHarness';
open_system(model);
```

To specify options for the optimization, such as the allowable word length and number of iterations, use the `fxpOptimizationOptions` object.

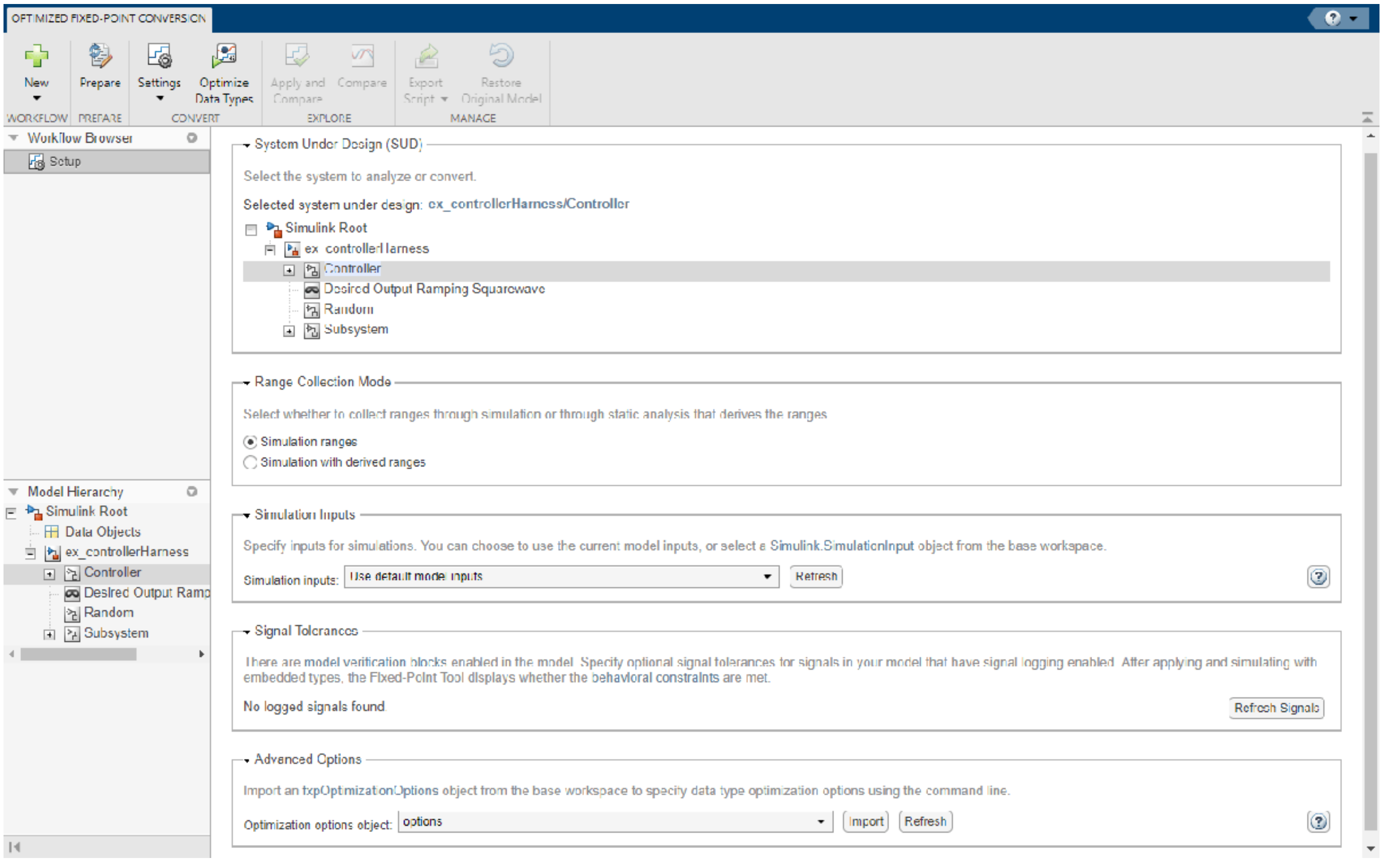
```
options = fxpOptimizationOptions('AllowableWordLengths',[2:32],...
    'MaxIterations',3e2,...
    'Patience',50);
```

Open the Fixed-Point Tool with the `Controller` subsystem selected.

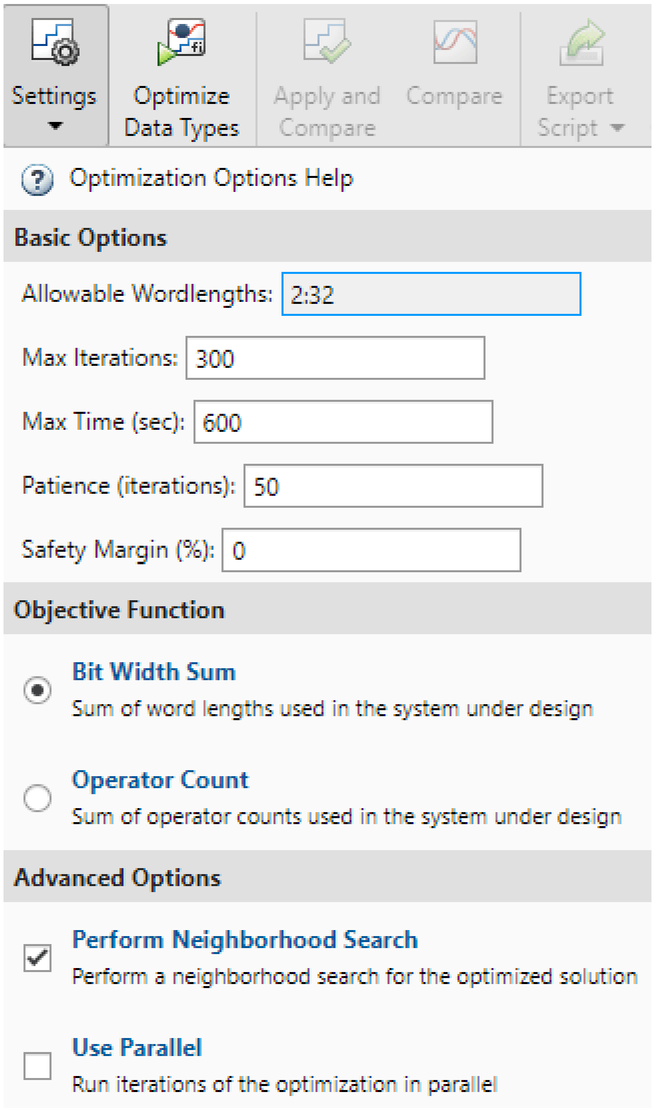
```
fxptdlg('ex_controllerHarness/Controller')
```

In the Fixed-Point Tool, select **New > Optimized Fixed-Point Conversion** to start the data type optimization workflow.

In the **Setup** pane, under **Advanced Options**, select the optimization options object to import from the dropdown menu. Click **Import**.



Expand the **Settings** menu in the toolstrip to confirm that the optimization options were applied.



Version History

Introduced in R2018a

R2021b: Restrict instrumentation to a subsystem

You can now restrict instrumentation to a subsystem by using the `InstrumentationContext` property of the `fxpOptimizationOptions` object to specify the subsystem to use for instrumentation and range collection.

R2021b: Warn about unsupported constructs

You can now choose to display a warning message when `fxpopt` encounters blocks that are not supported for data type conversion, in addition to the existing options to isolate or error. To warn for unsupported constructs, set the `HandleUnsupported` property of the `fxpOptimizationOptions` object to `'Warn'`.

R2021a: Override data types with scaled doubles

You can now override data types in a model with scaled doubles.

See Also

Classes
`OptimizationResult` | `OptimizationSolution`

Functions
`addTolerance` | `showTolerances` | `explore` | `fxpopt`

Topics
"Optimize Fixed-Point Data Types for a System"

OptimizationResult class

Result after optimizing fixed-point system

Description

An `OptimizationResult` object contains the results after optimizing a fixed-point system. If the optimization process succeeds in finding a new fixed-point implementation, you can use this object to explore the different implementations that met the specified tolerances found during the process. Use the `explore` method to open the Simulation Data Inspector and view the behavior of the optimized system.

Construction

`result = fxpopt(model, sud, options)` optimizes the data types in the system specified by `sud` in the model, `model`, with additional options specified in the `fxpOptimizationOptions` object, `options`.

Input Arguments

`model` — Model containing system under design
character vector

Name of the model containing the system that you want to optimize.

Data Types: `char`

`sud` — System whose data types you want to optimize
character vector

System whose data types you want to optimize, specified as a character vector containing the path to the system.

Data Types: `char`

`options` — Additional optimization options
`fxpOptimizationOptions` object

`fxpOptimizationOptions` object specifying additional options to use during the data type optimization process.

Properties

`FinalOutcome` — Message specifying whether a new optimal solution was found
character vector

Message specifying whether the optimization process found a new optimal solution, returned as a character vector.

Data Types: `char`

`OptimizationOptions` — `fxpOptimizationOptions` object associated with the result
`fxpOptimizationOptions` object

The `fxpOptimizationOptions` object used as an input to the `fxpopt` function used to generate the `OptimizationResult`.

`Solutions` — Vector of `OptimizationSolution` objects
`OptimizationSolution` object | vector of `OptimizationSolution` objects

A vector of `OptimizationSolution` objects found during the optimization process. If the optimization finds a feasible solution, the vector is sorted by cost, with the lowest cost (most optimal) solution as the first element of the vector. If the optimization does not find a feasible solution, the vector is sorted by maximum difference from the original design.

Methods

explore	Explore fixed-point implementations found during optimization process
revert	Revert system data types and settings changed during optimization to original state
openSimulationManager	Inspect simulations run during optimization in Simulation Manager

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Version History

Introduced in R2018a

See Also

Classes
`fxpOptimizationOptions` | `OptimizationSolution`

Functions
`addTolerance` | `showTolerances` | `explore` | `fxpopt`

Topics
"Optimize Fixed-Point Data Types for a System"

OptimizationSolution class

Optimized fixed-point implementation of system

Description

An `OptimizationSolution` object is a fixed-point implementation of a system whose data types were optimized using the `fxpopt` function.

Construction

`solution = explore(result)` opens the Simulation Data Inspector. If the optimization found a solution, it returns the `OptimizationSolution` object with the lowest cost out of the vector of `OptimizationSolution` objects contained in the `OptimizationResult` object, `result`. If the optimization did not find a solution, it returns the `OptimizationSolution` object with the smallest `MaxDifference`.

You can also access a `OptimizationSolution` object by indexing the `Solutions` property of an `OptimizationResult` object. For example, to access the solution with the second lowest cost contained in the `OptimizationResult` object, `result`, enter

```
solution = result.Solutions(2)
```

Input Arguments

`result` — `OptimizationResult` containing the solution
`OptimizationResult` object

The `Solutions` property of the `OptimizationResult` object is a vector of `OptimizationSolution` objects found during the optimization process. If the optimization found a feasible solution, the vector is sorted by cost, with the lowest cost (most optimal) solution as the first element of the vector. If the optimization did not find a feasible solution, the vector is sorted by `MaxDifference`, with the solution with the smallest `MaxDifference` as the first element.

Properties

`Cost` — Sum of word lengths used in the system under design
scalar integer

Sum of all word lengths used in the solution in the system under design. The most optimal solution is the solution with the smallest cost.

Data Types: `double`

`Pass` — Whether the solution meets specified criteria
1 | 0

Whether the solution meets the criteria specified by the associated `fxpOptimizationOptions` object, specified as a logical.

Data Types: `logical`

`MaxDifference` — Maximum absolute difference between baseline solution run
scalar

The maximum absolute difference between the baseline the solution.

Data Types: `double`

`RunID` — Run identifier
scalar integer

Unique numerical identification for the run used by the Simulation Data Inspector. For more information, see “Inspect and Compare Data Programmatically”.

Data Types: `double`

`RunName` — Name of the run
character vector

Name of the run in Simulation Data Inspector.

Data Types: `char`

Methods

showContents	Get summary of changes made during data type optimization

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Version History

Introduced in R2018a

See Also

Classes
`fxpOptimizationOptions` | `OptimizationResult`

Functions
`addTolerance` | `showTolerances` | `explore` | `fxpopt`

Topics
“Optimize Fixed-Point Data Types for a System”

6

Methods

isHeterogeneous

Class: `coder.CellType`
Package: `coder`

Determine whether cell array type represents a heterogeneous cell array

Syntax

`tf = isHeterogeneous(t)`

Description

`tf = isHeterogeneous(t)` returns `true` if the `coder.CellType` object `t` is heterogeneous. Otherwise, it returns `false`.

Examples

Determine Whether Cell Array Type Is Heterogeneous

Create a `coder.CellType` object for a cell array whose elements have different classes.

```
t = coder.typeof({'a', 1})
```

```
t = 

coder.CellType
   1x2 heterogeneous cell 
      f0: 1x1 char
      f1: 1x1 double
```

Determine whether the `coder.CellType` object represents a heterogeneous cell array.

```
isHeterogeneous(t)
```

```
ans =

     1
```

Tips

- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The makeHomogeneous method makes a homogeneous copy of a type. The makeHeterogeneous method makes a heterogeneous copy of a type.

The makeHomogeneous and makeHeterogeneous methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

Version History

Introduced in R2015b

See Also

coder.typeof | coder.newtype

Topics

"Code Generation for Cell Arrays"
"Specify Cell Array Inputs at the Command Line"

isHomogeneous

Class: coder.CellType
Package: coder

Determine whether cell array type represents a homogeneous cell array

Syntax

```
tf = isHomogeneous(t)
```

Description

`tf = isHomogeneous(t)` returns `true` if the `coder.CellType` object `t` represents a homogeneous cell array. Otherwise, it returns `false`.

Examples

Determine Whether Cell Array Type Is Homogeneous.

Create a `coder.CellType` object for a cell array whose elements have the same class and size.

```
t = coder.typeof({1 2 3})
```

```
t = 

coder.CellType
   1x3 homogeneous cell
      base: 1x1 double
```

Determine whether the `coder.CellType` object represents a homogeneous cell array.

```
isHomogeneous(t)
```

```
ans =

     1
```

Test for a Homogeneous Cell Array Type Before Executing Code

Write a function `make_varsize`. If the input type `t` is homogeneous, the function returns a variable-size copy of `t`.

```
function c = make_varsize(t, n)
assert(isHomogeneous(t));
c = coder.typeof(t, [n n], [1 1]);
end
```

Create a heterogeneous type `tc`.

```
tc = coder.typeof({'a', 1});
```

Pass `tc` to `make_varsize`.

```
tc1 = make_varsize(tc, 5)
```

The assertion fails because `tc` is heterogeneous.

Create a homogeneous type `tc`.

```
tc = coder.typeof({1 2 3});
```

Pass `tc` to `make_varsize`.

```
tc1 = make_varsize(tc, 5)
```

```
tc1 =

coder.CellType
   :5x:5 homogeneous cell
      base: 1x1 double
```

Tips

- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods. The `makeHomogeneous` method makes a homogeneous copy of a type. The `makeHeterogeneous` method makes a heterogeneous copy of a type.

  The `makeHomogeneous` and `makeHeterogeneous` methods permanently assign the classification as homogeneous and heterogeneous, respectively. You cannot later use one of these methods to create a copy that has a different classification.

Version History

Introduced in R2015b

See Also

`coder.typeof` | `coder.newtype`

Topics

"Code Generation for Cell Arrays"
"Specify Cell Array Inputs at the Command Line"

makeHeterogeneous

Class: coder.CellType
Package: coder

Make a heterogeneous copy of a cell array type

Syntax

```
newt = makeHeterogeneous(t)
t = makeHeterogeneous(t)
```

Description

`newt = makeHeterogeneous(t)` creates a `coder.CellType` object for a heterogeneous cell array from the `coder.CellType` object `t`. `t` cannot represent a variable-size cell array.

The classification as heterogeneous is permanent. You cannot later create a homogeneous `coder.CellType` object from `newt`.

`t = makeHeterogeneous(t)` creates a heterogeneous `coder.CellType` object from `t` and replaces `t` with the new object.

Examples

Replace a Homogeneous Cell Array Type with a Heterogeneous Cell Array Type

Create a cell array type `t` whose elements have the same class and size.

```
t = coder.typeof({1 2 3})
```

```
t = 

coder.CellType
   1x3 homogeneous cell
      base: 1x1 double
```

The cell array type is homogeneous.

Replace `t` with a cell array type for a heterogeneous cell array.

```
t = makeHeterogeneous(t)
```

```
t = 

coder.CellType
   1×3 locked heterogeneous cell
      f1: 1×1 double
      f2: 1×1 double
      f3: 1×1 doublee
```

The cell array type is heterogeneous. The elements have the size and class of the original homogeneous cell array type.

Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods.

Version History

Introduced in R2015b

See Also

`coder.typeof` | `coder.newtype`

Topics

"Code Generation for Cell Arrays"
"Specify Cell Array Inputs at the Command Line"

makeHomogeneous

Class: coder.CellType
Package: coder

Create a homogeneous copy of a cell array type

Syntax

```
newt = makeHomogeneous(t)
t = makeHomogeneous(t)
```

Description

`newt = makeHomogeneous(t)` creates a `coder.CellType` object for a homogeneous cell array `newt` from the `coder.CellType` object `t`.

To create `newt`, the `makeHomogeneous` method must determine a size and class that represent all elements of `t`:

- If the elements of `t` have the same class, but different sizes, the elements of `newt` are variable size with upper bounds that accommodate the elements of `t`.
- If the elements of `t` have different classes, for example, `char` and `double`, the `makeHomogeneous` method cannot create a `coder.CellType` object for a homogeneous cell array.

The classification as homogeneous is permanent. You cannot later create a heterogeneous `coder.CellType` object from `newt`.

`t = makeHomogeneous(t)` creates a homogeneous `coder.CellType` object from `t` and replaces `t` with the new object.

Examples

Replace a Heterogeneous Cell Array Type with a Homogeneous Cell Array Type

Create a cell array type `t` whose elements have the same class, but different sizes.

```
t = coder.typeof({1 [2 3]})
```

```
t =

coder.CellType
   1x2 heterogeneous cell
      f0: 1x1 double
      f1: 1x2 double
```

The cell array type is heterogeneous.

Replace `t` with a cell array type for a homogeneous cell array.

```
t = makeHomogeneous(t)
```

```
t = 

coder.CellType
   1×2 locked homogeneous cell
      base: 1×:2 double
```

The new cell array type is homogeneous.

Tips

- In the display of a `coder.CellType` object, the terms `locked heterogeneous` or `locked homogeneous` indicate that the classification as homogeneous or heterogeneous is permanent. You cannot later change the classification by using the `makeHomogeneous` or `makeHeterogeneous` methods.
- `coder.typeof` determines whether the cell array type is homogeneous or heterogeneous. If the cell array elements have the same class and size, `coder.typeof` returns a homogeneous cell array type. If the elements have different classes, `coder.typeof` returns a heterogeneous cell array type. For some cell arrays, the classification as homogeneous or heterogeneous is ambiguous. For example, the type for {1 [2 3]} can be a 1x2 heterogeneous type. The first element is double and the second element is 1x2 double. The type can also be a 1x3 homogeneous type in which the elements have class double and size 1x:2. For these ambiguous cases, `coder.typeof` uses heuristics to classify the type as homogeneous or heterogeneous. If you want a different classification, use the `makeHomogeneous` or `makeHeterogeneous` methods.

Version History

Introduced in R2015b

See Also

`coder.typeof` | `coder.newtype`

Topics

"Code Generation for Cell Arrays"
"Specify Cell Array Inputs at the Command Line"

addApproximation

Class: coder.FixPtConfig
Package: coder

Replace floating-point function with lookup table during fixed-point conversion

Syntax

addApproximation(approximationObject)

Description

addApproximation(approximationObject) specifies a lookup table replacement in a coder.FixPtConfig object. During floating-point to fixed-point conversion, the conversion process generates a lookup table approximation for the function specified in the approximationObject.

Input Arguments

approximationObject — Function replacement configuration object
coder.mathfcngenerator.LookupTable configuration object

Function replacement configuration object that specifies how to create an approximation for a MATLAB function. Use the coder.FixPtConfig configuration object addApproximation method to associate this configuration object with a coder.FixPtConfig object. Then use the fiaccel function -float2fixed option with coder.FixPtConfig to convert floating-point MATLAB code to fixed-point MATLAB code.

Examples

Replace log function with an optimized lookup table replacement

Create a function replacement configuration object that specifies to replace the log function with an optimized lookup table.

```
logAppx = coder.approximation('Function','log','OptimizeLUTSize',...
          true,'InputRange',[0.1,1000],'InterpolationDegree',1,...
          'ErrorThreshold',1e-3,...
          'FunctionNamePrefix','log_optim_','OptimizeIterations',25);
```

Create a fixed-point configuration object and associate the function replacement configuration object with it.

```
fixptcfg = coder.config('fixpt');
fixptcfg.addApproximation(logAppx);
```

You can now generate fixed-point code using the fiaccel function.

See Also

coder.FixPtConfig | fiaccel

Topics
“Replace the exp Function with a Lookup Table”
“Replace a Custom Function with a Lookup Table”
“Replacing Functions Using Lookup Table Approximations”

addDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Add design range specification to parameter

Syntax

`addDesignRangeSpecification(fcnName,paramName,designMin, designMax)`

Description

`addDesignRangeSpecification(fcnName,paramName,designMin, designMax)` specifies the minimum and maximum values allowed for the parameter, `paramName`, in function, `fcnName`. The fixed-point conversion process uses this design range information to derive ranges for downstream variables in the code.

Input Arguments

`fcnName` — Function name
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

`designMin` — Minimum value allowed for this parameter
scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

`designMax` — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

Add a Design Range Specification

Set up the fixed-point configuration object

```
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
```

Derive ranges and generate fixed-point code

```
fiaccel -float2fixed cfg dti;
```

See Also

coder.FixPtConfig | fiaccel | hasDesignRangeSpecification | removeDesignRangeSpecification | clearDesignRangeSpecifications | getDesignRangeSpecification

addFunctionReplacement

Class: coder.FixPtConfig
Package: coder

Replace floating-point function with fixed-point function during fixed-point conversion

Syntax

addFunctionReplacement(floatFn,fixedFn)

Description

addFunctionReplacement(floatFn,fixedFn) specifies a function replacement in a coder.FixPtConfig object. During floating-point to fixed-point conversion, the conversion process replaces the specified floating-point function with the specified fixed-point function. The fixed-point function must be in the same folder as the floating-point function or on the MATLAB path.

Input Arguments

floatFn — Name of floating-point function
'' (default) | string

Name of floating-point function, specified as a string.

fixedFn — Name of fixed-point function
'' (default) | string

Name of fixed-point function, specified as a string.

Examples

Specify Function Replacement in Fixed-Point Conversion Configuration Object

Suppose that:

- The function myfunc calls a local function myadd.
- The test function mytest calls myfunc.
- You want to replace calls to myadd with the fixed-point function fi_myadd.

Create a coder.FixPtConfig object, fixptcfg, with default settings.

```
fixptcfg = coder.config('fixpt');
```

Set the test bench name. In this example, the test bench function name is mytest.

```
fixptcfg.TestBenchName = 'mytest';
```

Specify that the floating-point function, myadd, should be replaced with the fixed-point function, fi_myadd.

```
fixptcfg.addFunctionReplacement('myadd', 'fi_myadd');
```

Convert the floating-point MATLAB function, `myfunc`, to fixed-point.

```
fiaccel -float2fixed fixptcfg myfunc
```

`fiaccel` replaces `myadd` with `fi_myadd` during floating-point to fixed-point conversion.

See Also

`coder.FixPtConfig` | `fiaccel`

addFunctionReplacement

Class: `coder.SingleConfig`
Package: `coder`

Replace double-precision function with single-precision function during single-precision conversion

Syntax

`addFunctionReplacement(doubleFn,singleFn)`

Description

`addFunctionReplacement(doubleFn,singleFn)` specifies a function replacement in a `coder.SingleConfig` object. During double-precision to single-precision conversion, the conversion process replaces the specified double-precision function with the specified single-precision function. The single-precision function must be in the same folder as the double-precision function or on the MATLAB path. It is a best practice to provide unique names to local functions that a replacement function calls. If a replacement function calls a local function, do not give that local function the same name as a local function in a different replacement function file.

Input Arguments

`doubleFn` — Name of double-precision function
`''` (default) | string

Name of double-precision function, specified as a string.

`singleFn` — Name of single-precision function
`''` (default) | string

Name of single-precision function, specified as a string.

Examples

Specify Function Replacement in Single-Precision Conversion Configuration Object

Suppose that:

- The function `myfunc` calls a local function `myadd`.
- The test function `mytest` calls `myfunc`.
- You want to replace calls to `myadd` with the single-precision function `single_myadd`.

Create a `coder.SingleConfig` object, `scfg`, with default settings.

```
scfg = coder.config('single');
```

Set the test file name. In this example, the test file function name is `mytest`.

```
scfg.TestBenchName = 'mytest';
```

Specify that you want to replace the double-precision function, `myadd`, with the single-precision function, `single_myadd`.

```
scfg.addFunctionReplacement('myadd', 'single_myadd');
```

Convert the double-precision MATLAB function, `myfunc` to a single-precision MATLAB function.

```
convertToSingle -config scfg myfunc
```

The double-precision to single-precision conversion replaces instances of `myadd` with `single_myadd`.

Version History

Introduced in R2015b

clearDesignRangeSpecifications

Class: `coder.FixPtConfig`
Package: `coder`

Clear all design range specifications

Syntax

`clearDesignRangeSpecifications()`

Description

`clearDesignRangeSpecifications()` clears all design range specifications.

Examples

Clear a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now remove the design range
cfg.clearDesignRangeSpecifications()
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

See Also

`coder.FixPtConfig` | `fiaccel` | `addDesignRangeSpecification` | `removeDesignRangeSpecification` | `hasDesignRangeSpecification` | `getDesignRangeSpecification`

getDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Get design range specifications for parameter

Syntax

```
[designMin, designMax] = getDesignRangeSpecification(fcnName,paramName)
```

Description

`[designMin, designMax] = getDesignRangeSpecification(fcnName,paramName)` gets the minimum and maximum values specified for the parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Function name
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

Output Arguments

`designMin` — Minimum value allowed for this parameter
scalar

Minimum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

`designMax` — Maximum value allowed for this parameter
scalar

Maximum value allowed for this parameter, specified as a scalar double.

Data Types: `double`

Examples

Get Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Get the  design range for the 'dti' function parameter 'u_in'
[designMin, designMax] = cfg.getDesignRangeSpecification('dti','u_in')
```

```
designMin =

    -1


designMax =

     1
```

See Also

coder.FixPtConfig | fiaccel | addDesignRangeSpecification | hasDesignRangeSpecification | removeDesignRangeSpecification | clearDesignRangeSpecifications

hasDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Determine whether parameter has design range

Syntax

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)`

Description

`hasDesignRange = hasDesignRangeSpecification(fcnName,paramName)` returns true if the parameter, `param_name` in function, `fcn`, has a design range specified.

Input Arguments

`fcnName` — Name of function
string

Function name, specified as a string.

Example: ‘dti’

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Example: ‘dti’

Data Types: `char`

Output Arguments

`hasDesignRange` — Parameter has design range
true | false

Parameter has design range, returned as a boolean.

Data Types: `logical`

Examples

Verify That a Parameter Has a Design Range Specification

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
```

```
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0);
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

```
hasDesignRanges =

     1
```

See Also

coder.FixPtConfig | fiaccel | addDesignRangeSpecification | removeDesignRangeSpecification | clearDesignRangeSpecifications | getDesignRangeSpecification

removeDesignRangeSpecification

Class: `coder.FixPtConfig`
Package: `coder`

Remove design range specification from parameter

Syntax

`removeDesignRangeSpecification(fcnName,paramName)`

Description

`removeDesignRangeSpecification(fcnName,paramName)` removes the design range information specified for parameter, `paramName`, in function, `fcnName`.

Input Arguments

`fcnName` — Name of function
string

Function name, specified as a string.

Data Types: `char`

`paramName` — Parameter name
string

Parameter name, specified as a string.

Data Types: `char`

Examples

Remove Design Range Specifications

```
% Set up the fixed-point configuration object
cfg = coder.config('fixpt');
cfg.TestBenchName = 'dti_test';
cfg.addDesignRangeSpecification('dti', 'u_in', -1.0, 1.0)
cfg.ComputeDerivedRanges = true;
% Verify that the 'dti' function parameter 'u_in' has design range
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
% Now clear the design ranges and verify that
% hasDesignRangeSpecification returns false
cfg.removeDesignRangeSpecification('dti', 'u_in')
hasDesignRanges = cfg.hasDesignRangeSpecification('dti','u_in')
```

See Also

coder.FixPtConfig | fiaccel | addDesignRangeSpecification | clearDesignRangeSpecifications | hasDesignRangeSpecification | getDesignRangeSpecification

applyDataTypes

Package: `DataTypeWorkflow`

Apply proposed data types to model

Syntax

`applyDataTypes(converter,RunName)`

Description

`applyDataTypes(converter,RunName)` applies the proposed data types for the specified run, `RunName`, to the system specified by the `converter` object.

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

`RunName` — Name of run to apply data types to
character vector

Name of run to apply data types to, specified as a character vector.

Example: `applyDataTypes(converter,'Run1')`

Data Types: `char`

Alternatives

The `applyDataTypes` object function provides functionality similar to the Fixed-Point Tool button **Apply Data Types** . For more information, see Fixed-Point Tool.

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.ProposalSettings` | `proposeDataTypes`

Topics
"Convert a Model to Fixed Point Using the Command Line"

applySettingsFromRun

Package: `DataTypeWorkflow`

Apply system settings used in previous run to model

Syntax

`applySettingsFromRun(converter,RunName)`

Description

`applySettingsFromRun(converter,RunName)` applies the data type override and instrumentation settings used in a previous run, `RunName`, to the model specified in the `converter` object.

Input Arguments

`converter` — Converter object

`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

`RunName` — Name of run

character vector

Name of run from which to apply settings, specified as a character vector.

Example: `applySettingsFromRun(converter,'Run1')`

Data Types: `char`

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.Converter` | `applySettingsFromShortcut`

Topics

"Convert a Model to Fixed Point Using the Command Line"

applySettingsFromShortcut

Package: `DataTypeWorkflow`

Apply settings from shortcut to model

Syntax

```
applySettingsFromShortcut(converter,shortcutName)
```

Description

`applySettingsFromShortcut(converter,shortcutName)` applies settings from the specified system shortcut, `shortcutName`, to a `converter` object.

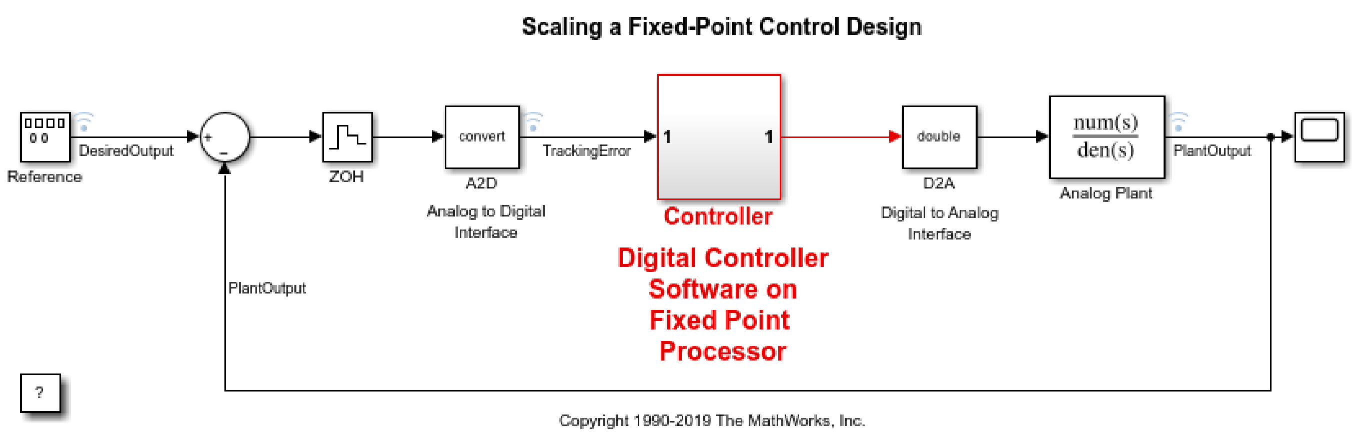
Examples

Configure Model for Conversion Using a Shortcut

This example shows how to configure a model for fixed-point conversion using a shortcut.

Open the `fxpdemo_feedback` model.

```
open_system('fxpdemo_feedback');
```



Create a `DataTypeWorkflow.Converter` object for the Controller subsystem.

```
converter = DataTypeWorkflow.Converter('fxpdemo_feedback/Controller');
```

Configure the model for conversion by using a shortcut. Find the shortcuts that are available for the system by accessing the `ShortcutsForSelectedSystem` property of the `converter` object.

```
shortcuts = converter.ShortcutsForSelectedSystem
```

```
shortcuts =
```

```
  6x1 cell array

    {'Range collection using double override'       }
    {'Range collection with specified data types'   }
    {'Range collection using single override'       }
    {'Disable range collection'                     }
    {'Remove overrides and disable range collection'}
    {'Range collection using scaled double override'}
```

To collect idealized ranges for the system, use the `'Range collection using double override'` shortcut to override the system with double-precision data types and enable instrumentation.

```
applySettingsFromShortcut(converter,shortcuts{1})
```

This shortcut also updates the current run name property of the converter object.

```
converter.CurrentRunName
```

```
ans =

    'Ranges(Double)'
```

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

`shortcutName` — Name of shortcut
character vector

Name of the shortcut that specifies which settings to use, specified as a character vector.

Example: `applySettingsFromShortcut(converter,'Range collection using double override')`

Data Types: `char`

Version History

Introduced in R2014b

See Also

`applySettingsFromRun` | `DataTypeWorkflow.Converter`

Topics
"Convert a Model to Fixed Point Using the Command Line"

deriveMinMax

Package: `DataTypeWorkflow`

Derive range information for model

Syntax

`deriveMinMax(converter)`

Description

`deriveMinMax(converter)` derives the minimum and maximum values for each block in the system specified by the `DataTypeWorkflow.Converter` object based on design minimum and maximum values.

Input Arguments

`converter` — Converter object for system under design
`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

Tips

If any issues come up during the derivation, they can be queried using the `proposalIssues` object function.

Alternatives

The `deriveMinMax` object function is equivalent to the **Collect Ranges** button () with **Range Collection Mode** set to **Derived Ranges** in the Fixed-Point Tool. For more information, see Fixed-Point Tool.

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.Converter` | `simulateSystem` | `proposalIssues`

Topics
"Convert a Model to Fixed Point Using the Command Line"

proposeDataTypes

Package: `DataTypeWorkflow`

Propose data types for system

Syntax

`proposeDataTypes(converter,RunName,propSettings)`

Description

`proposeDataTypes(converter,RunName,propSettings)` proposes data types for the system specified by the `DataTypeWorkflow.Converter` object, `converter`, based on the range results stored in `RunName` and the settings specified in `propSettings`.

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object, specified as a `DataTypeWorkflow.Converter` object, for the system under design.

`RunName` — Name of run
character vector

Name of run to propose data types for, specified as a character vector.

Data Types: `char`

`propSettings` — Proposed data type settings
`DataTypeWorkflow.ProposalSettings` object

Proposed data type settings, specified as a `DataTypeWorkflow.ProposalSettings` object. Use this object to specify proposal settings such as the default data type for all floating point signals.

Data Types: `char`

Alternatives

The `proposeDataTypes` object function provides functionality similar to the Fixed-Point Tool **Propose Data Types**  button. For more information, see Fixed-Point Tool.

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.Converter` | `DataTypeWorkflow.ProposalSettings` | `applyDataTypes`

Topics

"Convert a Model to Fixed Point Using the Command Line"

results

Package: `DataTypeWorkflow`

Find results for selected system in converter object

Syntax

```
results = results(converter,RunName)
results = results(converter,RunName,filterFunc)
```

Description

`results = results(converter,RunName)` returns all results in the specified run, for the model specified by the `DataTypeWorkflow.Converter` object, `converter`.

`results = results(converter,RunName,filterFunc)` returns the results in the specified run that match the criteria specified by `filterFunc`.

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

`RunName` — Name of run
character vector

Name of the run to query, specified as a character vector.

Data Types: `char`

`filterFunc` — Function to use to filter results
function handle

Function to use to filter results, specified as a function handle with a `DataTypeWorkflow.Result` object as its input.

Data Types: `function_handle`

Output Arguments

`results` — Filtered results
array of `Result` objects

Filtered results, returned as an array of `DataTypeWorkflow.Result` objects.

Alternatives

The `results` object function offers a command-line approach to using the Fixed-Point Tool. For more information, see Fixed-Point Tool.

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.Converter` | `proposalIssues` | `wrapOverflows` | `saturationOverflows`

Topics

"Convert a Model to Fixed Point Using the Command Line"

proposalIssues

Package: `DataTypeWorkflow`

Get results which have comments associated with them

Syntax

`results = proposalIssues(converter,RunName)`

Description

`results = proposalIssues(converter,RunName)` returns all results in `RunName` for the model specified by a `DataTypeWorkflow.Converter` object, `converter`, that have associated comments. The comments field of the returned results provides information related to any issues found.

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object for system under design, specified as a `DataTypeWorkflow.Converter` object.

`RunName` — Name of run
character vector

Name of the run to look for comments in, specified as a character vector.

Data Types: `char`

Output Arguments

`results` — Results that have associated comments
`DataTypeWorkflow.Result` object

Results that have associated comments, returned as a `DataTypeWorkflow.Result` object, for all signals in `RunName`.

Alternatives

The `DataTypeWorkflow.Converter.proposalIssues` object function offers a command-line approach to using the Fixed-Point Tool. See Fixed-Point Tool for more information.

Version History

Introduced in R2014b

See Also

`DataTypeWorkflow.Converter` | `results` | `wrapOverflows` | `saturationOverflows`

Topics
"Convert a Model to Fixed Point Using the Command Line"

saturationOverflows

Package: DataTypeWorkflow

Get results where saturation occurred

Syntax

```
results = saturationOverflows(converter,RunName)
```

Description

`results = saturationOverflows(converter,RunName)` returns all results in `RunName`, for the model specified by the `DataTypeWorkflow.Converter` object, `converter`, that saturated during simulation.

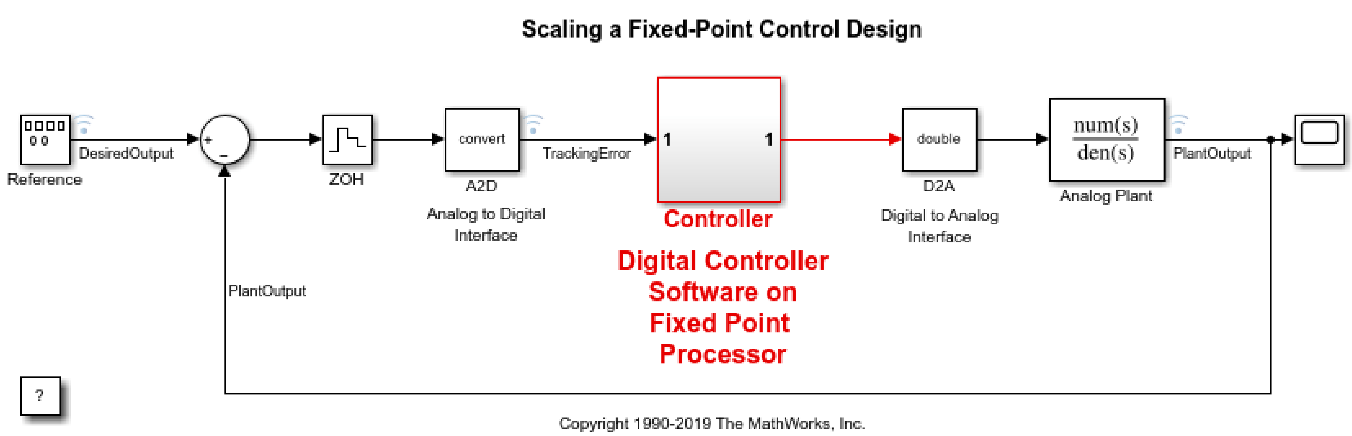
Examples

Get Saturation Results for Specified Run

This example shows how to get saturation results for the specified run of a `DataTypeWorkflow.Converter` object.

Open the `fxpdemo_feedback` model.

```
open_system('fxpdemo_feedback');
```



Create a `DataTypeWorkflow.Converter` object for the Controller subsystem.

```
converter = DataTypeWorkflow.Converter('fxpdemo_feedback/Controller');
```

Simulate the model and store the results in a run titled `InitialRun`.

```
converter.CurrentRunName = 'InitialRun';
simulateSystem(converter);
```

Determine if there were any overflows in the run.

```
saturations = saturationOverflows(converter,'InitialRun')
```

```
saturations = 

  Result with properties:

           ResultName: 'fxpdemo_feedback/Controller/Up Cast'
    SpecifiedDataType: 'fixdt(1,16,14)'
     CompiledDataType: 'fixdt(1,16,14)'
     ProposedDataType: ''
                Wraps: []
          Saturations: 23
          WholeNumber: 0
               SimMin: -2
               SimMax: 1.9999
           DerivedMin: []
           DerivedMax: []
              RunName: 'InitialRun'
             Comments: {'An output data type cannot be specified on this result. The output type
            DesignMin: []
            DesignMax: []
```

A saturation occurs in the Up Cast block of the Controller subsystem during the simulation. There are no wrapping overflows.

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

`RunName` — Name of run
character vector

Name of run to look for saturations in, specified as a character vector.

Example: `saturations = saturationOverflows(converter,'Run 1')`

Data Types: `char`

Output Arguments

`results` — Results that saturated
`DataTypeWorkflow.Result` object

Results that saturated, returned as a `DataTypeWorkflow.Result` object.

Version History
Introduced in R2014b

See Also

`DataTypeWorkflow.Converter` | `results` | `wrapOverflows` | `proposalIssues`

Topics

"Convert a Model to Fixed Point Using the Command Line"

simulateSystem

Package: DataTypeWorkflow

Simulate system specified by converter object

Syntax

```
simOut = simulateSystem(converter)
simOut = simulateSystem(converter,Name,Value)
simOut = simulateSystem(converter,simIn)
simOut = simulateSystem(converter,ParameterStruct)
simOut = simulateSystem(converter,ConfigSet)
```

Description

`simOut = simulateSystem(converter)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`.

`simOut = simulateSystem(converter,Name,Value)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`, using additional options specified by one or more `Name,Value` pair arguments. This function accepts the same `Name,Value` pairs as the `sim` function.

`simOut = simulateSystem(converter,simIn)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`, using the inputs specified in the `Simulink.SimulationInput` object `simIn`.

`simOut = simulateSystem(converter,ParameterStruct)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`, using the parameter values specified in the structure, `ParameterStruct`.

`simOut = simulateSystem(converter,ConfigSet)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`, using the configuration settings specified in the model configuration set, `ConfigSet`.

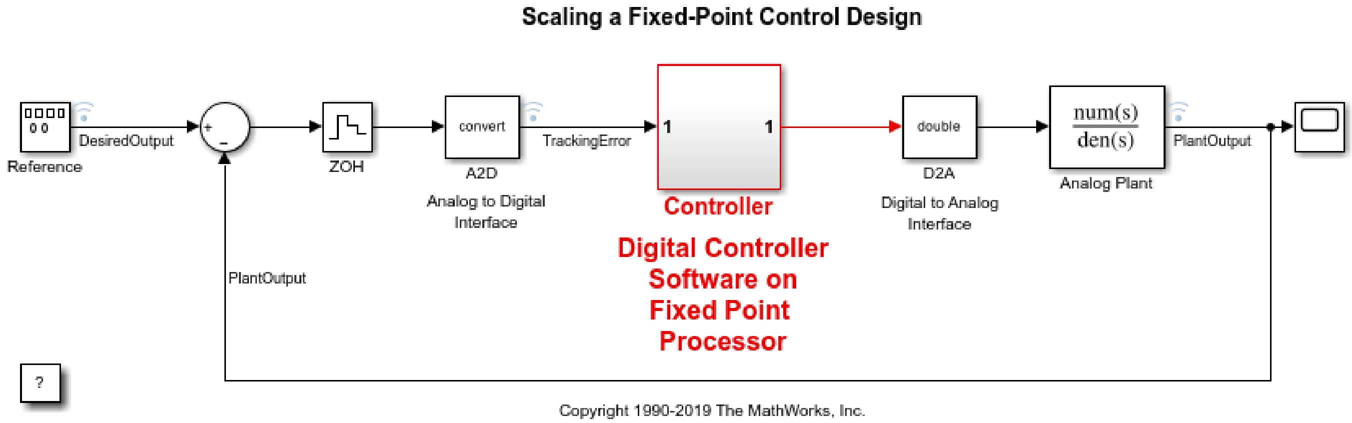
Examples

Simulate a `DataTypeWorkflow.Converter` Object's System

This example shows how to simulate the `converter` object's system.

Open the `fxpdemo_feedback` model.

```
open_system('fxpdemo_feedback');
```



Create a `DataTypeWorkflow.Converter` object for the Controller subsystem.

```
converter = DataTypeWorkflow.Converter('fxpdemo_feedback/Controller');
```

Simulate the model.

```
simulateSystem(converter);
```

Input Arguments

`converter` — Converter object

`DataTypeWorkflow.Converter` object

Converter object for the system under design, specified as a `DataTypeWorkflow.Converter` object.

`simIn` — Simulation input for the system

`Simulink.SimulationInput` object | array of `Simulink.SimulationInput` objects

Simulation input for the system, specified as a `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects.

When you use a `SimulationInput` object as an input to the `simulateSystem` function, you can also specify the following `Name,Value` pair arguments.

Parameter	Values
`ShowSimulationManager`	• `'on'` - Opens the **Simulation Manager**. • `'off'` (default) - Does not open the Simulation Manager.
`ShowProgress`	• `'on'` - View the progress of the simulation in the command window. • `'off'` (default) - The progress of the simulation does not display in the command window.

`ParameterStruct` — Parameter settings

structure

Names of the configuration parameters for the simulation, specified as a structure. The corresponding values are the parameter values.

Data Types: `struct`

`ConfigSet` — Configuration set
`Simulink.ConfigSet` object

Configuration set, specified as a `Simulink.ConfigSet` object, that contains the values of the model parameters.

Output Arguments

`simOut` — Simulation output
`Simulink.SimulationOutput` object

Simulation output, returned as a `Simulink.SimulationOutput` object. The returned object includes the simulation outputs: logged time, states, and signals.

Tips

- To name your simulation run, before simulation, change the `CurrentRunName` property of the `DataTypeWorkflow.Converter` object.
- `simulateSystem` provides functionality similar to the `sim` command, except that `simulateSystem` preserves the model-wide data type override and instrumentation settings of each run.

Note

- The `SimulationMode` property must be set to `normal`. The Fixed-Point Designer software does collect simulation ranges in Rapid accelerator or Hot restart modes.
- The `StopTime` property cannot be set to `inf`.
- The `SrcWorkspace` parameter must be set to either `base` or `current`.

Version History

Introduced in R2014b

See Also

`sim` | `DataTypeWorkflow.Converter`

Topics

"Convert a Model to Fixed Point Using the Command Line"

verify

Package: `DataTypeWorkflow`

Compare behavior of baseline and autoscaled systems

Syntax

```
verificationResult = verify(converter,baselineRun,verificationRunName)
```

Description

`verificationResult = verify(converter,baselineRun,verificationRunName)` simulates the system specified by the `DataTypeWorkflow.Converter` object, `converter`, and stores the run information in a new run, `verificationRun`. It returns a `DataTypeWorkflow.VerificationResult` object that compares the baseline and verification runs.

Input Arguments

`converter` — Converter object for system to verify
`DataTypeWorkflow.Converter` object

Converter object for system to verify, specified as a `DataTypeWorkflow.Converter` object. The `DataTypeWorkflow.Converter` object contains instrumentation data from the baseline run, as well as the tolerances specified on the associated `DataTypeWorkflow.ProposalSettings` object. The software determines if the behavior of the verification run is acceptable using the tolerances specified on the `ProposalSettings` object.

`baselineRun` — Baseline run to compare against
character vector

Baseline run to compare against, specified as a character vector.

Data Types: `char` | `string`

`verificationRunName` — Name of the verification run to create
character vector

Name of the verification run to create during the embedded simulation, specified as a character vector.

Data Types: `char` | `string`

Output Arguments

`verificationResult` — Comparison of the baseline run and the verification run
`DataTypeWorkflow.VerificationResult` object

Comparison of the baseline run and the verification run, returned as a `DataTypeWorkflow.VerificationResult` object.

Version History

Introduced in R2019a

See Also

`DataTypeWorkflow.Converter` | `DataTypeWorkflow.ProposalSettings` | `DataTypeWorkflow.VerificationResult`

Topics

"Convert a Model to Fixed Point Using the Command Line"

wrapOverflows

Package: DataTypeWorkflow

Get results where wrapping occurred

Syntax

```
results = wrapOverflows(converter,RunName)
```

Description

`results = wrapOverflows(converter,RunName)` returns all results in `RunName`, for the system specified by the `DataTypeWorkflow.Converter` object, `converter`, that wrapped during simulation.

Input Arguments

`converter` — Converter object
`DataTypeWorkflow.Converter` object

Converter object, specified as a `DataTypeWorkflow.Converter` object, for the system under design.

`RunName` — Name of run
character vector

Name of run in which to look for wrap overflows, specified as a character vector.

Example: `results = wrapOverflows(converter,'Run3')`

Data Types: `char`

Output Arguments

`results` — Signals that wrapped during the specified run
`DataTypeWorkflow.Result` object

Signals that wrapped during the specified run, returned as a `DataTypeWorkflow.Result` object.

Version History

Introduced in R2014b

See Also

`results` | `saturationOverflows` | `proposalIssues`

Topics
"Convert a Model to Fixed Point Using the Command Line"

addTolerance

Package: DataTypeWorkflow

Specify numeric tolerance for converted system

Syntax

```
addTolerance(proposalSettings,block_path,port_index,tolerance_type,
tolerance_value)
```

Description

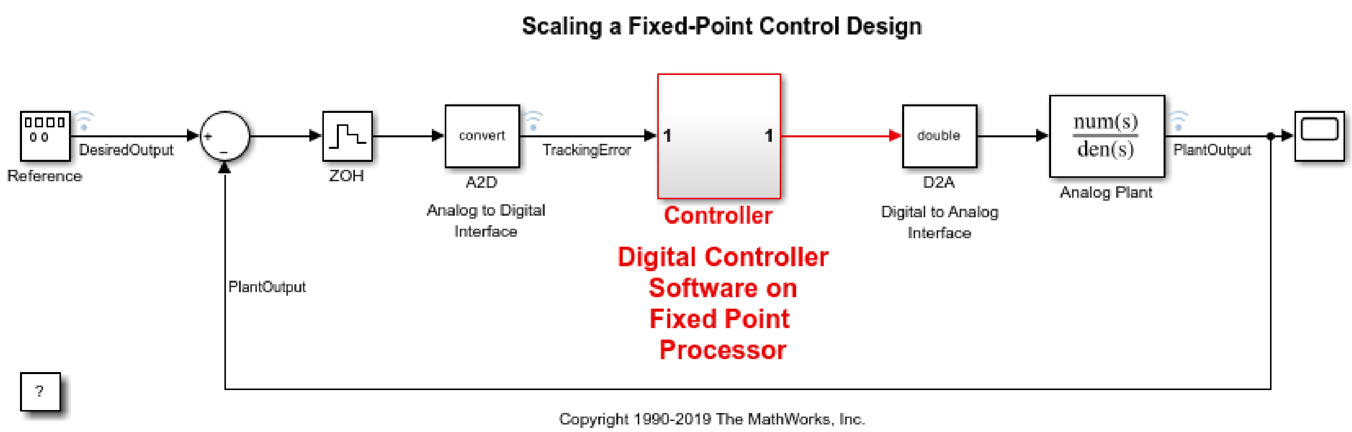
`addTolerance(proposalSettings,block_path,port_index,tolerance_type, tolerance_value)` adds numeric tolerance data to a `DataTypeWorkflow.ProposalSettings` object for the output signal specified by `block_path` and `port_index`, with the tolerance type specified by `tolerance_type` and value specified by `tolerance_value`.

Examples

Specify Signal Tolerances

This example shows how to apply and remove tolerances from signals in a system. In this example, you add tolerances to a `DataTypeWorkflow.proposalSettings` object, and then remove all tolerances from this object.

```
model = 'fxpdemo_feedback';
open_system(model);
```



Create a `DataTypeWorkflow.ProposalSettings` object.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

Add an absolute tolerance of 0.05 to the output of the Down Cast block in the Controller subsystem.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast',1,'AbsTol',5e-2);
```

Add a relative tolerance of 1% to the same signal.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast',1,'RelTol',1e-2);
```

Use `showTolerances` to see all tolerances associated with the proposal settings object.

```
showTolerances(propSettings)
```

```
                    Path                        Port_Index    Tolerance_Type    Tolerance_Value
    _________________________________________    __________    ______________    _______________

    {'fxpdemo_feedback/Controller/Down Cast'}        1           {'AbsTol'}           0.05
    {'fxpdemo_feedback/Controller/Down Cast'}        1           {'RelTol'}           0.01
```

Clear the tolerances stored in the `ProposalSettings` object.

```
clearTolerances(propSettings)
```

Using `showTolerances`, verify that there are no longer any tolerances stored in the `ProposalSettings` object.

```
showTolerances(propSettings)
```

Input Arguments

`proposalSettings` — Object that contains proposal settings
`DataTypeWorkflow.ProposalSettings` object

Object that contains proposal settings, specified as a `DataTypeWorkflow.ProposalSettings` object. You add tolerance specifications to the `DataTypeWorkflow.ProposalSettings` object.

`block_path` — Path to block for which to add tolerance
character vector

Path to the block for which to add a tolerance to, specified as a character vector.

Data Types: `char` | `string`

`port_index` — Index of output port of block
scalar integer

Index of the output port of the blocks, specified as a scalar integer.

Data Types: `double`

`tolerance_type` — Type of tolerance
`'AbsTol'` | `'RelTol'` | `'TimeTol'`

Type of tolerance, specified as one of these values:

- `'AbsTol'` – Absolute tolerance
- `'RelTol'` – Relative tolerance
- `'TimeTol'` – Time tolerance

Data Types: `char`

`tolerance_value` — Acceptable difference between original output and output of new design
scalar double

Acceptable difference between the original output and the output of the new design, specified as a scalar double.

If `tolerance_type` is set to `'AbsTol'`, then `tolerance_value` represents the absolute value of the maximum acceptable difference between the original output and the output of the new design.

If `tolerance_type` is set to `'RelTol'`, then `tolerance_value` represents the maximum relative difference, specified as a percentage, between the original output and the output of the new design. For example, a value of `1e-2` indicates a maximum difference of one percent between the original output and the output of the new design.

If `tolerance_type` is set to `'TimeTol'`, then `tolerance_value` defines a time interval, in seconds, in which the maximum and minimum values define the upper and lower values to compare against. For more information, see "How the Simulation Data Inspector Compares Data".

Data Types: `double`

Version History

Introduced in R2019a

See Also

`DataTypeWorkflow.ProposalSettings` | `showTolerances` | `clearTolerances`

Topics

"Convert a Model to Fixed Point Using the Command Line"
"The Command-Line Interface for the Fixed-Point Tool"

clearTolerances

Package: `DataTypeWorkflow`

Clear all tolerances specified by a `DataTypeWorkflow.ProposalSettings` object

Syntax

```
clearTolerances(proposalSettings)
```

Description

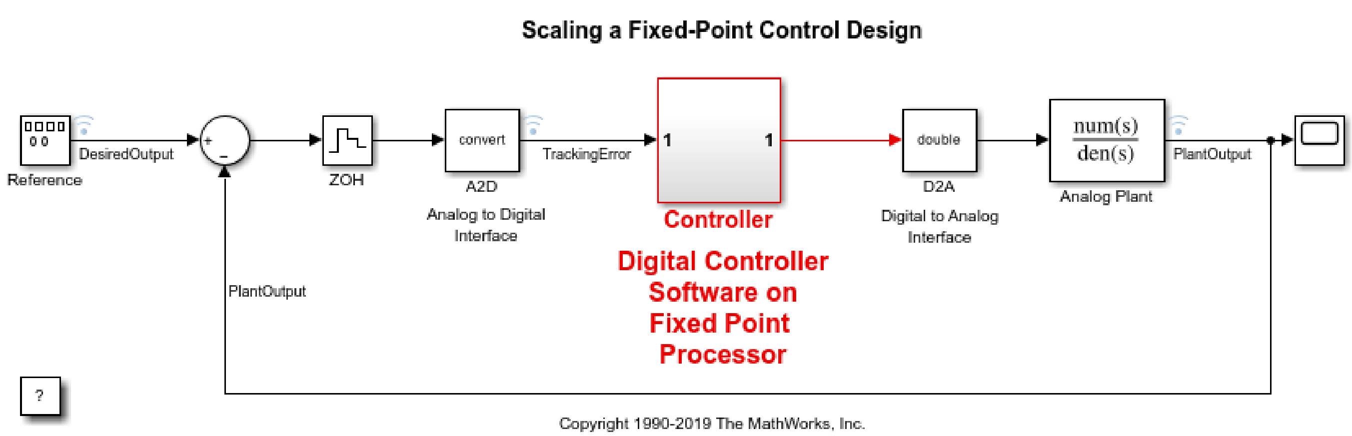
`clearTolerances(proposalSettings)` clears the absolute, relative, and time tolerances of a `proposalSettings` object.

Examples

Specify Signal Tolerances

This example shows how to apply and remove tolerances from signals in a system. In this example, you add tolerances to a `DataTypeWorkflow.proposalSettings` object, and then remove all tolerances from this object.

```
model = 'fxpdemo_feedback';
open_system(model);
```



Create a `DataTypeWorkflow.ProposalSettings` object.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

Add an absolute tolerance of 0.05 to the output of the Down Cast block in the Controller subsystem.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast',1,'AbsTol',5e-2);
```

Add a relative tolerance of 1% to the same signal.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast',1,'RelTol',1e-2);
```

Use `showTolerances` to see all tolerances associated with the proposal settings object.

```
showTolerances(propSettings)
```

```
                   Path                      Port_Index    Tolerance_Type    Tolerance_Value
    _________________________________________    __________    ______________    _______________

    {'fxpdemo_feedback/Controller/Down Cast'}        1            {'AbsTol'}           0.05
    {'fxpdemo_feedback/Controller/Down Cast'}        1            {'RelTol'}           0.01
```

Clear the tolerances stored in the `ProposalSettings` object.

```
clearTolerances(propSettings)
```

Using `showTolerances`, verify that there are no longer any tolerances stored in the `ProposalSettings` object.

```
showTolerances(propSettings)
```

Input Arguments

`proposalSettings` — Object that contains proposal settings
`DataTypeWorkflow.ProposalSettings` object

Object that contains proposal settings, specified as a `DataTypeWorkflow.ProposalSettings` object. A `DataTypeWorkflow.ProposalSettings` object specifies tolerances and settings to use during the data type proposal process.

Version History

Introduced in R2019a

See Also

`DataTypeWorkflow.ProposalSettings` | `showTolerances` | `addTolerance`

Topics

"Convert a Model to Fixed Point Using the Command Line"
"The Command-Line Interface for the Fixed-Point Tool"

showTolerances

Package: `DataTypeWorkflow`

Show tolerances specified for a system

Syntax

```
showTolerances(proposalSettings)
```

Description

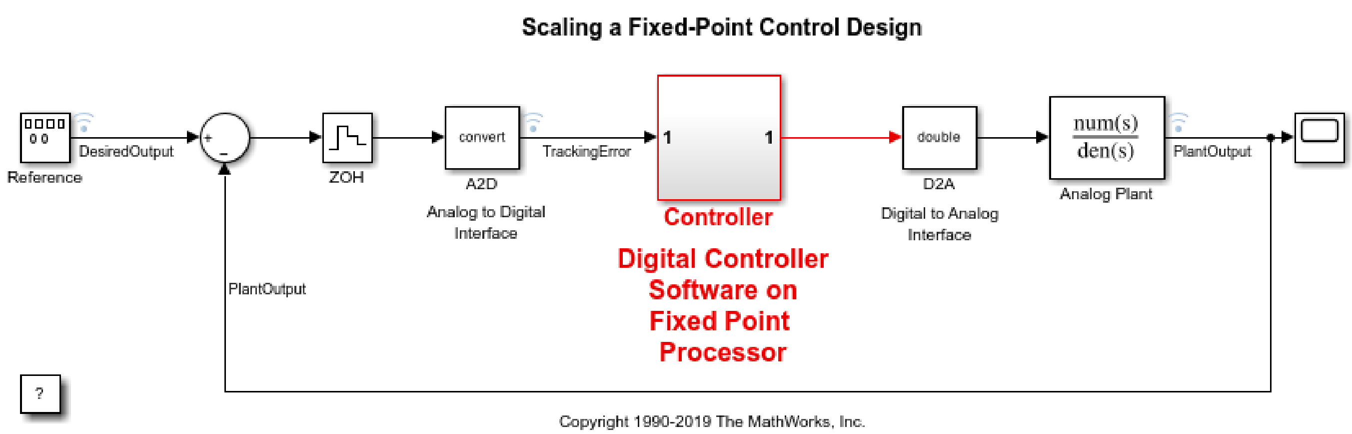
`showTolerances(proposalSettings)` displays the absolute, relative, and time tolerances specified for a system specified by the `proposalSettings` object. If the `proposalSettings` object has no tolerances specified, the `showTolerances` object function does not display anything.

Examples

Specify Signal Tolerances

This example shows how to apply and remove tolerances from signals in a system. In this example, you add tolerances to a `DataTypeWorkflow.proposalSettings` object, and then remove all tolerances from this object.

```
model = 'fxpdemo_feedback';
open_system(model);
```



Create a `DataTypeWorkflow.ProposalSettings` object.

```
propSettings = DataTypeWorkflow.ProposalSettings;
```

Add an absolute tolerance of 0.05 to the output of the Down Cast block in the Controller subsystem.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast',1,'AbsTol',5e-2);
```

Add a relative tolerance of 1% to the same signal.

```
addTolerance(propSettings, 'fxpdemo_feedback/Controller/Down Cast',1,'RelTol',1e-2);
```

Use `showTolerances` to see all tolerances associated with the proposal settings object.

```
showTolerances(propSettings)
```

```
                     Path                       Port_Index    Tolerance_Type    Tolerance_Value
    ________________________________________    __________    ______________    _______________

    {'fxpdemo_feedback/Controller/Down Cast'}       1           {'AbsTol'}           0.05
    {'fxpdemo_feedback/Controller/Down Cast'}       1           {'RelTol'}           0.01
```

Clear the tolerances stored in the `ProposalSettings` object.

```
clearTolerances(propSettings)
```

Using `showTolerances`, verify that there are no longer any tolerances stored in the `ProposalSettings` object.

```
showTolerances(propSettings)
```

Input Arguments

`proposalSettings` — Object that contains proposal settings
`DataTypeWorkflow.ProposalSettings` object

Object that contains proposal settings, specified as a `DataTypeWorkflow.ProposalSettings` object. This object specifies tolerances and settings to use during the data type proposal process.

Version History

Introduced in R2019a

See Also

`DataTypeWorkflow.ProposalSettings` | `clearTolerances` | `addTolerance`

Topics

“Convert a Model to Fixed Point Using the Command Line”
“The Command-Line Interface for the Fixed-Point Tool”

convertToSingle

Package: DataTypeWorkflow

Convert a double-precision system to single precision

Syntax

```
ConversionReport = DataTypeWorkflow.Single.convertToSingle(systemToConvert)
```

Description

ConversionReport = DataTypeWorkflow.Single.convertToSingle(systemToConvert) converts the system specified by systemToConvert to single precision and returns a report. Data types that are specified as Boolean, fixed point, or one of the built-in integers are not affected by conversion.

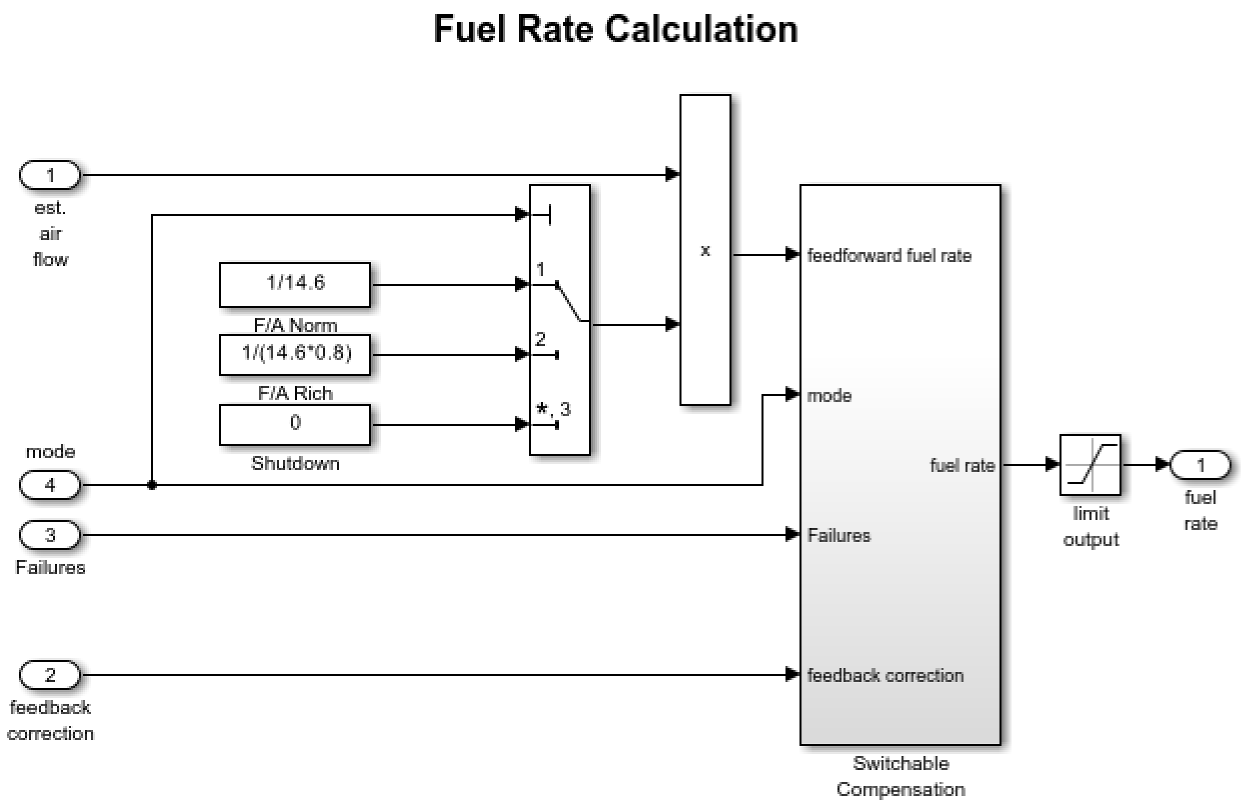
Examples

Convert System to Single Precision

This example shows how to convert a double-precision system to single precision.

Open the system to convert to single precision.

```
open_system('ex_fuel_rate_calculation');
```



Use the `DataTypeWorkflow.Single.convertToSingle` method to convert the system from double precision to single precision.

```
report = DataTypeWorkflow.Single.convertToSingle('ex_fuel_rate_calculation');
```

```
Updating Model Advisor cache...
Model Advisor cache updated. For new customizations, to update the cache, use the Advisor.Manage
```

The specified system now uses single-precision data types instead of double-precision data types. Data types in the model that were specified as Boolean, fixed-point, or one of the built-in integers remain the same after conversion.

Input Arguments

`systemToConvert` — System to convert to single precision

character vector

The system to convert from double-precision to single-precision, specified as a character vector. The system must be open before using this method.

Data Types: `char`

Output Arguments

`ConversionReport` — Report containing results from the conversion

report

Report containing results from the conversion.

Alternatives

You can also use the Single Precision Converter app to convert a system from double precision to single precision. To open the Single Precision Converter app, in the Simulink **Apps** tab, select **Single Precision Converter**. For more information, see “Getting Started with Single Precision Converter”.

Version History

Introduced in R2016b

See Also

Single Precision Converter

Topics

“Convert a System to Single Precision”
“Getting Started with Single Precision Converter”

explore

Package: DataTypeWorkflow

Explore comparison of baseline and fixed-point implementations

Syntax

explore(verificationResult)

Description

explore(verificationResult) opens the Simulation Data Inspector with the logged data for the DataTypeWorkflow.VerificationResult object specified by verificationResult.

Input Arguments

verificationResult — Object comparing behavior of a baseline run and a verification run
DataTypeWorkflow.VerificationResult object

Object comparing the behavior of a baseline run and a verification run, specified as a DataTypeWorkflow.VerificationResult object.

Version History

Introduced in R2019a

See Also

DataTypeWorkflow.Converter | DataTypeWorkflow.ProposalSettings | DataTypeWorkflow.VerificationResult

Topics

"Convert a Model to Fixed Point Using the Command Line"

getNumDataPointsInfo

Package: `fixed`

Get information about number of data points in generated data

Syntax

`datainfo = getNumDataPointsInfo(datagenerator)`

Description

`datainfo = getNumDataPointsInfo(datagenerator)` returns information about the data points generated by the `fixed.DataGenerator` object, `datagenerator`.

Examples

Get information about number of data points in generated data

The `getNumDataPointsInfo` function returns information related to the number of data points in the data generated from a `fixed.DataGenerator` object.

```
dataspec = fixed.DataSpecification('fixdt(1,16,13)',...
 'Intervals', {-1,1})
```

```
dataspec = 

  fixed.DataSpecification with properties:

          DataTypeStr: 'sfix16_En13'
            Intervals: [-1,1]
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

```
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...
 'NumDataPointsLimit', 20);
getNumDataPointsInfo(datagen)
```

```
ans = 

  struct with fields:

    Current: 20
       Next: 21
        Min: 5
        Max: 75
```

The output indicates that there are currently 20 data combinations in the generated data. The maximum number of combinations that the `DataGenerator` object would produce is 75.

Get information about number of data points for multidimensional data

When the dimension of the generated data is greater than one, it can be useful to find the next possible size of generated data.

Create a `DataGenerator` object where the associated `DataSpecification` object specifies 2-dimensional data.

```
dataspec = fixed.DataSpecification('single', 'Dimensions', 2);
datagen = fixed.DataGenerator('DataSpecifications', dataspec)
```

```
datagen = 

  fixed.DataGenerator with properties:

    DataSpecifications: {[1×1 fixed.DataSpecification]}
    NumDataPointsLimit: 100000
```

The `DataGenerator` object uses the default limit of 100000 data points in the generated data.

Get information about the number of data points generated.

```
getNumDataPointsInfo(datagen)
```

```
ans = 

  struct with fields:

    Current: 99856
       Next: 100489
        Min: 81
        Max: 130321
```

The current size of the generated data is 99856 points. By setting the `NumDataPointsLimit` property of the `DataGenerator` object to the value specified in `Max`, you can get the maximum possible number of data combinations.

Set the `NumDataPointsLimit` property of the `DataGenerator` object to the maximum possible number of data points.

```
datagen.NumDataPointsLimit = 130321;
getNumDataPointsInfo(datagen)
```

```
ans = 

  struct with fields:

    Current: 130321
       Next: 130321
        Min: 81
        Max: 130321
```

Input Arguments

`datagenerator` — Object from which you want to get information
`fixed.DataGenerator` object

Object from which you want to get information, specified as a `fixed.DataGenerator` object.

Output Arguments

`datainfo` — Information about the number of data points
struct

Information about the number of data points in the data generated from a `fixed.DataGenerator` object, returned as a struct with the following fields.

Field	Description
`Current`	The number of data combinations in the generated data.
`Next`	Next possible size of data combinations.
`Min`	Minimum number of combinations of data required to be in the generated data. This number is equal to the number of boundary values and mandatory values in the `DataSpecification` objects associated with the `DataGenerator` object.
`Max`	Maximum number of combinations that could be in the generated data.

Version History

Introduced in R2019b

See Also

`fixed.DataGenerator` | `getUniqueValues` | `outputAllData`

getUniqueValues

Package: `fixed`

Get unique values from `fixed.DataGenerator` object

Syntax

`data = getUniqueValues(datagenerator)`

Description

`data = getUniqueValues(datagenerator)` returns all unique values in the data generated by the `fixed.DataGenerator` object, `datagenerator`.

Examples

Get unique values in data from `DataGenerator` object

In data generated from a `fixed.DataGenerator` object, there can be repeated values. Use the `getUniqueValues` function to get all of the unique values in the data set.

```
dataspec = fixed.DataSpecification('fixdt(1,16,13)',...
 'Intervals', {-1,1})
```

```
dataspec = 

  fixed.DataSpecification with properties:

          DataTypeStr: 'sfix16_En13'
            Intervals: [-1,1]
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

```
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...
 'NumDataPointsLimit', 20);
getUniqueValues(datagen)
```

```
ans = 

   -1.0000
   -0.9999
   -0.4999
   -0.2500
   -0.0624
   -0.0313
   -0.0039
   -0.0021
   -0.0005
   -0.0002
         0
```

```
    0.0010
    0.0018
    0.0078
    0.0155
    0.0157
    0.1249
    0.1251
    0.9999
    1.0000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13
```

Input Arguments

`datagenerator` — Input `fixed.DataGenerator` object
`fixed.DataGenerator` object

Input `fixed.DataGenerator` object to get unique values from.

Output Arguments

`data` — Unique set of values in data
scalar | vector | matrix

Unique set of data generated by the input `fixed.DataGenerator` object, returned as a scalar, vector, or matrix.

Version History

Introduced in R2019b

See Also

`fixed.DataGenerator` | `getNumDataPointsInfo` | `outputAllData`

outputAllData

Package: fixed

Get data from fixed.DataGenerator object

Syntax

```
data = outputAllData(datagenerator)
data = outputAllData(datagenerator, format)
```

Description

data = outputAllData(datagenerator) returns the data generated by the fixed.DataGenerator object, datagenerator.

data = outputAllData(datagenerator, format) returns the data generated by the fixed.DataGenerator object, datagenerator, in the format specified by format.

Examples

Get data as an array

Get the data from a fixed.DataGenerator object, returned as an array of values.

```
dataspec = fixed.DataSpecification('int8', 'Intervals', {-1,1});
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...
 'NumDataPointsLimit', 20)
```

```
datagen = 

  fixed.DataGenerator with properties:

    DataSpecifications: {[1×1 fixed.DataSpecification]}
    NumDataPointsLimit: 20
```

Use the outputAllData function to access the data in the DataGenerator object.

```
data = outputAllData(datagen)
```

```
data =

  1×3 int8 row vector

   -1    0    1
```

The function returns the data in an array with the type specified by the fixed.DataSpecification object.

Get data as a timeseries object

Get the data from a `fixed.DataGenerator` object, returned as a `timeseries` object.

```
dataspec = fixed.DataSpecification('int8', 'Intervals', {-1,1});
datagen = fixed.DataGenerator('DataSpecifications', dataspec,...
 'NumDataPointsLimit', 2000)
```

```
datagen = 

  fixed.DataGenerator with properties:

    DataSpecifications: {[1×1 fixed.DataSpecification]}
    NumDataPointsLimit: 20000
```

Specify the format of the output type to get a `timeseries` object.

```
data = outputAllData(datagen, 'timeseries')
```

```
  timeseries

  Common Properties:
            Name: 'unnamed'
            Time: [3x1 double]
        TimeInfo: [1x1 tsdata.timemetadata]
            Data: [3x1 int8]
        DataInfo: [1x1 tsdata.datametadata]
```

Input Arguments

`datagenerator` — Object from which you want to get data
`fixed.DataGenerator` object

Object from which you want to get data, specified as a `fixed.DataGenerator` object.

`format` — Format in which you want data returned
`'array'` (default) | `'timeseries'` | `'dataset'`

Format in which you want data returned, specified as either `'array'`, `'timeseries'`, or `'dataset'`.

Specifying `'dataset'` returns a `Simulink.SimulationData.Dataset` object. Specifying `'timeseries'` returns a `timeseries` object.

Example: `data = outputAllData(datagen, 'timeseries');`

Data Types: `char`

Output Arguments

`data` — Data from the `DataGenerator` object
scalar | vector | matrix | `timeseries` object

Data from the `DataGenerator` object, returned as either a scalar, vector, matrix, or `timeseries` object.

Version History

Introduced in R2019b

See Also

`fixed.DataGenerator` | `getUniqueValues` | `getNumDataPointsInfo`

applyOnRootInport

Package: `fixed`

(To be removed) Apply properties to Inport block

Note `applyOnRootInport` will be removed in a future release.

Syntax

```
applyOnRootInport(dataspec, model, inportnumber)
```

Description

`applyOnRootInport(dataspec, model, inportnumber)` applies the properties specified in `fixed.DataSpecification` object, `dataspec` to the specified Inport block in `model`.

Input Arguments

`dataspec` — Properties to apply to Inport block
`fixed.DataSpecification` object

Properties to apply to Inport block, specified as a `fixed.DataSpecification` object.

`model` — Model containing Inport block
character vector

Name of the model containing the Inport block to apply settings to, specified as a character vector.

Data Types: `char`

`inportnumber` — Number of Inport block
scalar integer

Port number of root-level Inport block on which you want to apply properties from the `fixed.DataSpecification` object. The following properties of the `DataSpecification` object are applied to the block:

- Data type
- Complexity
- Dimensions

Data Types: `double`

Version History

Introduced in R2019b

R2020a: `applyOnRootInport` will be removed
Warns starting in R2020a

`applyOnRootInport` will be removed in a future release.

See Also

`fixed.DataSpecification` | `contains`

contains

Package: `fixed`

Determine whether value domain of a `DataSpecification` object contains a specified value

Syntax

`bool = contains(dataspec, value)`

Description

`bool = contains(dataspec, value)` returns a boolean value indicating whether the value domain of the `fixed.DataSpecification` object, `dataspec`, contains the value, `value`.

Examples

Determine whether a `fixed.DataSpecification` object contains a value

Use the `contains` function to determine whether a `fixed.DataSpecification` object contains a specified value.

```
dataspec = fixed.DataSpecification('int8', 'Intervals', {-1,1})
```

```
dataspec = 

  fixed.DataSpecification with properties:

          DataTypeStr: 'int8'
            Intervals: [-1,1]
      MandatoryValues: <empty>
           Complexity: 'real'
           Dimensions: 1
```

Determine whether `dataspec` contains the value 0.

```
bool = contains(dataspec, 0)
```

```
bool = 

  logical

   1
```

Input Arguments

`dataspec` — `fixed.DataSpecification` object
`fixed.DataSpecification` object

Input `fixed.DataSpecification` object.

`value` — Value
scalar | vector

Value or values to check for in the `fixed.DataSpecification` object, specified as a scalar, or vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

Output Arguments

`bool` — Whether the `fixed.DataSpecification` object contains the value
true | false | vector of logical values

Whether the `fixed.DataSpecification` object contains the value, returned as a boolean value.

If the `value` argument is a vector, the output is a boolean vector of the same length.

Version History

Introduced in R2019b

See Also

`fixed.DataSpecification` | `applyOnRootInport`

contains

Package: fixed

Determine if one fixed.Interval object contains another

Syntax

```
bool = contains(A, B)
```

Description

bool = contains(A, B) returns a boolean indicating whether fixed.Interval object A contains the fixed.Interval object B.

Examples

Determine if a fixed.Interval object contains another

Create two fixed.Interval objects. Use the contains function to determine if the intervals in interval2 are contained within the corresponding intervals in interval1.

```
interval1 = fixed.Interval({0,1}, {2,3}, {3,4});
interval2 = fixed.Interval({0,0.5}, {2.5, 3}, {4,5});
bool = contains(interval1, interval2)
```

```
bool = 1x3 logical array

   1   1   0
```

When the second input is a scalar Interval object, contains determines whether each interval of the first input contains the interval of the second input.

```
interval2 = fixed.Interval(0,1);
bool = contains(interval1, interval2)
```

```
bool = 1x3 logical array

   1   0   0
```

Input Arguments

A, B — Input fixed.Interval objects
fixed.Interval object | array of fixed.Interval objects

Input fixed.Interval objects, specified as fixed.Interval objects, or arrays of fixed.Interval objects.

If `A` is an array of `Interval` objects, `B` must be a scalar `Interval` object or an `Interval` object with the same dimensions as `A`.

Output Arguments

`bool` — Whether `B` is contained in `A`
true | false | logical array

Whether `fixed.Interval` object `B` is contained in `fixed.Interval` object `A`, returned as a logical value.

When `A` is an array of `Interval` objects, the output is an array of logical values of the same size as `A`.

Version History

Introduced in R2019b

See Also

`fixed.Interval` | `intersect` | `overlaps` | `setdiff` | `union` | `unique`

intersect

Package: `fixed`

Intersection of `fixed.Interval` objects

Syntax

`C = intersect(A, B)`

Description

`C = intersect(A, B)` returns the intersection of `fixed.Interval` objects `A` and `B`.

Examples

Get intersection of two `fixed.Interval` objects

Create two `fixed.Interval` objects.

```
interval1 = fixed.Interval(-10,10)
```

```
interval1 = 
    [-10,10]

  1x1 fixed.Interval with properties:

              LeftEnd: -10
             RightEnd: 10
         IsLeftClosed: true
        IsRightClosed: true
```

```
interval2 = fixed.Interval(0,20)
```

```
interval2 = 
    [0,20]

  1x1 fixed.Interval with properties:

              LeftEnd: 0
             RightEnd: 20
         IsLeftClosed: true
        IsRightClosed: true
```

Find the intersection of the two `Interval` objects.

```
intervalIntersection12 = intersect(interval1,interval2)
```

```
intervalIntersection12 = 
    [0,10]

  1x1 fixed.Interval with properties:
```

```
              LeftEnd: 0
             RightEnd: 10
         IsLeftClosed: true
        IsRightClosed: true
```

The output is an `Interval` object whose range is the intersection of the ranges of the two input `Interval` objects.

When the ranges of the two input `Interval` objects do not overlap, the output is an empty `Interval` object.

```
interval3 = fixed.Interval(100,200)
```

```
interval3 = 
    [100,200]

  1x1 fixed.Interval with properties:

              LeftEnd: 100
             RightEnd: 200
         IsLeftClosed: true
        IsRightClosed: true
```

```
intervalIntersection13 = intersect(interval1,interval3)
```

```
intervalIntersection13 = 

  1x0 fixed.Interval with properties:

              LeftEnd
             RightEnd
         IsLeftClosed
        IsRightClosed
```

Input Arguments

A, B — Input fixed.Interval objects
fixed.Interval object | array of fixed.Interval objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

Output Arguments

C — Intersection of fixed.Interval objects
fixed.Interval object | array of fixed.Interval objects

Intersection of input `fixed.Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

The output `Interval` object contains all values in both inputs, `A` and `B`.

Version History
Introduced in R2019b

See Also
fixed.Interval | contains | overlaps | setdiff | union | unique

isDegenerate

Package: `fixed`

Determine whether the left and right ends of a `fixed.Interval` object are degenerate

Syntax

`bool = isDegenerate(A)`

Description

`bool = isDegenerate(A)` returns a boolean indicating whether the left and right ends of the `fixed.Interval` object `A` are the same, or equivalently, whether the interval contains only one point.

Examples

Determine if a `fixed.Interval` object has degenerate end points

Create a `fixed.Interval` object. Use the `isDegenerate` function to determine whether the left and right ends of the `Interval` object are the same.

```
interval = fixed.Interval({-pi,pi},{1,1});
bool = isDegenerate(interval)
```

```
bool = 1x2 logical array

   0   1
```

The output is a logical `0` when the left and right ends of the interval are different, and `1` when they are the same.

Input Arguments

`A` — `fixed.Interval` object
`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

Output Arguments

`bool` — Indicates whether left and right ends of *A* are degenerate
true | false | logical array

Indicates whether the `fixed.Interval` object `A` has degenerate end points. Returns `1` (true) when the left and right ends of `A` are the same, or equivalently, when the interval contains only one point, and `0` (false) otherwise.

When `A` is an array of `Interval` objects, the output is an array of logical values of the same size as `A`.

Version History

Introduced in R2019b

See Also

`isLeftBounded` | `isRightBounded` | `isnan` | `fixed.Interval`

isLeftBounded

Package: `fixed`

Determine whether a `fixed.Interval` object is left-bounded

Syntax

`bool = isLeftBounded(A)`

Description

`bool = isLeftBounded(A)` returns a boolean indicating whether the `fixed.Interval` object `A` is left-bounded.

Examples

Determine if a `fixed.Interval` object is left bounded

Create a `fixed.Interval` object. Use the `isLeftBounded` function to determine whether the interval is bounded on the left.

```
interval = fixed.Interval({-pi,pi},{-inf,1});
bool = isLeftBounded(interval)
```

```
bool = 1x2 logical array

   1   0
```

The output is a logical `1` when the left end of the interval is bounded, and `0` otherwise.

Input Arguments

`A` — `fixed.Interval` object
`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

Output Arguments

`bool` — Indicates whether left end of *A* is bounded
true | false | logical array

Indicates whether the `fixed.Interval` object `A` is left-bounded, returned as a logical value. Returns `0` (false) when `A` contains `-inf`, and `1` (true) otherwise.

When `A` is an array of `Interval` objects, the output is an array of logical values of the same size as `A`.

Version History

Introduced in R2019b

See Also

`isDegenerate` | `isRightBounded` | `isnan` | `fixed.Interval`

isnan

Package: fixed

Determine whether a fixed.Interval object is NaN

Syntax

```
bool = isnan(A)
```

Description

bool = isnan(A) returns a boolean indicating whether a fixed.Interval object A is NaN.

Examples

Determine if a fixed.Interval object is NaN

Create a fixed.Interval object. Use the isnan function to determine whether the Interval object is not a number.

```
interval = fixed.Interval({-pi,pi},{nan,1},{nan,nan});
bool = isnan(interval)
```

```
bool = 1x3 logical array

   0   1   1
```

The output is a logical 1 when the interval contains one or more NaN elements, and 0 otherwise.

Input Arguments

A — fixed.Interval object
fixed.Interval object | array of fixed.Interval objects

Input fixed.Interval object, specified as a fixed.Interval object, or an array of fixed.Interval objects.

Output Arguments

bool — Indicates whether elements of *A* are NaN
true | false | logical array

Indicates whether the fixed.Interval object A is NaN, returned as a logical value.

When A is an array of Interval objects, the output is an array of logical values of the same size as A.

Version History

Introduced in R2019b

See Also

`isDegenerate` | `isLeftBounded` | `isRightBounded` | `fixed.Interval`

isRightBounded

Package: `fixed`

Determine whether the a `fixed.Interval` object is right-bounded

Syntax

`bool = isRightBounded(A)`

Description

`bool = isRightBounded(A)` returns a boolean indicating whether the `fixed.Interval` object `A` is right-bounded.

Examples

Determine if a `fixed.Interval` object is right bounded

Create a `fixed.Interval` object. Use the `isRightBounded` function to determine whether the interval is bounded on the right.

```
interval = fixed.Interval({-pi,pi},{-1,inf});
bool = isRightBounded(interval)
```

```
bool = 1x2 logical array

   1   0
```

The output is logical `1` when the right end of the interval is bounded, and `0` otherwise.

Input Arguments

`A` — `fixed.Interval` object
`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

Output Arguments

`bool` — Indicates whether right end of *A* is bounded
`Boolean` scalar | `Boolean` array

Indicates whether the `fixed.Interval` object `A` is right-bounded, returned as a logical value. Returns `0` (false) when `A` contains `inf`, and `1` (true) otherwise.

When `A` is an array of `Interval` objects, the output is an array of logical values of the same size as `A`.

Version History

Introduced in R2019b

See Also

`isDegenerate` | `isLeftBounded` | `isnan` | `fixed.Interval`

overlaps

Package: `fixed`

Determine if two `fixed.Interval` objects overlap

Syntax

```
bool = overlaps(A, B)
```

Description

`bool = overlaps(A, B)` returns a boolean indicating whether two `fixed.Interval` objects overlap.

Examples

Determine if two `fixed.Interval` objects overlap

Create two `fixed.Interval` objects and determine if their ranges overlap.

```
interval1 = fixed.Interval(-1, 1);
interval2 = fixed.Interval(0, 1);
overlaps(interval1, interval2)
```

```
ans =

  logical

   1
```

When the ranges of the `Interval` objects overlap, the `overlaps` function returns a value of 1, or `true`.

Input Arguments

`A, B` — Input `fixed.Interval` objects
`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

Output Arguments

`bool` — Whether the intervals overlap
true | false | vector of logical values

Whether the input `fixed.Interval` objects overlap, returned as a logical value or a vector of logical values.

Version History

Introduced in R2019b

See Also

`fixed.Interval` | `contains` | `intersect` | `setdiff` | `union` | `unique`

quantize

Package: fixed

Quantize interval to range of numeric data type

Syntax

```
quantizedinterval = quantize(interval, numerictype)
quantizedinterval = quantize(interval, numerictype, Name,Value)
```

Description

`quantizedinterval = quantize(interval, numerictype)` returns the quantized range of `fixed.Interval` object, `interval`, quantized to the numeric type specified by `numerictype`.

`quantizedinterval = quantize(interval, numerictype, Name,Value)` returns the quantized range of `fixed.Interval` object, `interval`, with additional properties specified as name-value pairs.

Examples

Quantize a numeric interval to uint8

Create a `fixed.Interval` object and find the range of the `Interval` object quantized to an unsigned 8-bit integer.

```
interval = fixed.Interval(-200,200);
quantizedInterval = quantize(interval, 'fixdt(0,8,0)')
```

```
quantizedInterval =

  1×2 uint8 row vector

     0   200
```

Because `fixdt(0,8,0)` is equivalent to `uint8`, the `quantize` function returns the quantized range as a `uint8` row vector with the endpoints within the representable range of the numeric type.

To return the quantized row vector as a fixed-point data type, set the `'PreferBuiltIn'` property to `false`.

```
quantizedInterval = quantize(interval, 'fixdt(0,8,0)',...
 'PreferBuiltIn', false)
```

```
quantizedInterval =

     0   200

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Unsigned
```

```
        WordLength: 8
    FractionLength: 0
```

Input Arguments

`interval` — Input `fixed.Interval` objects to quantize

`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

`numerictype` — Numeric data type

`Simulink.Numerictype` object | `embedded.numerictype` object | character vector

Numeric data type to quantize the `Interval`, specified as a `Simulink.Numerictype` object, an `embedded.numerictype` object, or a character vector representing a numeric data type, for example, `'single'`.

Example: `quantizedinterval = quantize(interval, 'fixdt(1,16,8)');`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `interval = quantize(interval, 'fixdt(1,16,0)', 'PreferBuiltIn', false, 'PreferStrict', true);`

`PreferBuiltIn` — Quantize to built-in data type when possible

true (default) | false

When this property is true, if the specified `numerictype` has an equivalent built-in integer type the software returns the built-in type. For example, when this property is `true`, a specified `numerictype` of `'fixdt(1,8,0)'` would return an `int8`.

Data Types: `logical`

`PreferStrict` — Quantize end points to numeric type

false (default) | true

When this property is true, all ends are quantized to the closest representable values within original intervals regardless of whether the intervals are closed or open.

Data Types: `logical`

Output Arguments

`quantizedinterval` — Quantized interval range

N-by-2 matrix

N-by-2 matrix with rows consisting of endpoints of input `Interval` objects quantized to the numeric data type specified by `numerictype`.

When the `'PreferStrict'` property is set to `false`, the end points after quantization may lie outside the original interval.

Version History

Introduced in R2019b

See Also

`fixed.Interval` | `contains` | `intersect` | `overlaps` | `union` | `unique`

setdiff

Package: `fixed`

Set difference of `fixed.Interval` objects

Syntax

```
C = setdiff(A, B)
```

Description

`C = setdiff(A, B)` returns a `fixed.Interval` object containing the values in `fixed.Interval` object `A`, but not in `B`.

Examples

Get set difference of two `fixed.Interval` objects

Create two `fixed.Interval` objects. Use the `setdiff` function to find the values that are in `Interval` object `interval1` but not in `interval2`. In this example, `interval1` contains all values between 0 and 1, but `interval2` only contains values from 0 to 0.5, so the output `Interval` object has an interval from 0.5 to 1.

```
interval1 = fixed.Interval(0,1);
interval2 = fixed.Interval(0,0.5);
intervaldiff = setdiff(interval1, interval2)
```

```
intervaldiff = 
    (0.5000,1]

  1x1 fixed.Interval with properties:

              LeftEnd: 0.5000
             RightEnd: 1
         IsLeftClosed: false
        IsRightClosed: true
```

Create an interval object that excludes zero

You can use the `setdiff` function to create an interval object based on another interval, while excluding zero.

Create an `Interval` object that contains zero.

```
myInterval = fixed.Interval(-1,1);
```

To create an interval based on the Interval object, myInterval, use the setdiff function. Include the constructor for a degenerate Interval object containing only zero as the second argument.

```
myInterval_nozero = setdiff(myInterval, {0});
```

```
myInterval_nozero = 

    [-1,0)    (0,1]

  1x2 fixed.Interval with properties:

              LeftEnd
             RightEnd
         IsLeftClosed
        IsRightClosed
```

The output Interval object, myInterval_nozero, contains two intervals, each with an open end point at zero. Therefore, the interval contains all values between -1 and 1, except 0.

Input Arguments

A, B — Input fixed.Interval objects
fixed.Interval object | array of fixed.Interval objects

Input fixed.Interval objects, specified as fixed.Interval objects, or arrays of fixed.Interval objects.

Output Arguments

C — Set difference of fixed.Interval objects
fixed.Interval object | array of fixed.Interval objects

Set difference of input fixed.Interval objects, returned as a fixed.Interval object or an array of fixed.Interval objects.

The output Interval object contains all values in first input, A, but not in B.

Version History

Introduced in R2019b

See Also

fixed.Interval | contains | intersect | overlaps | union

union

Package: `fixed`

Union of `fixed.Interval` objects

Syntax

`C = union(A, B)`

Description

`C = union(A, B)` returns the union of `fixed.Interval` objects `A` and `B`.

Examples

Get the union of two `fixed.Interval` objects

Create two `fixed.Interval` objects.

```
interval1 = fixed.Interval(-10, 10)
```

```
interval1 = 
    [-10,10]

  1x1 fixed.Interval with properties:

              LeftEnd: -10
             RightEnd: 10
         IsLeftClosed: true
        IsRightClosed: true
```

```
interval2 = fixed.Interval(0,20)
```

```
interval2 = 
    [0,20]

  1x1 fixed.Interval with properties:

              LeftEnd: 0
             RightEnd: 20
         IsLeftClosed: true
        IsRightClosed: true
```

Find the union of the two `Interval` objects.

```
intervalUnion = union(interval1, interval2)
```

```
intervalUnion = 
    [-10,20]

  1x1 fixed.Interval with properties:
```

```
             LeftEnd: -10
            RightEnd: 20
        IsLeftClosed: true
       IsRightClosed: true
```

The output is an `Interval` object whose range is the union of the ranges of the two input objects.

When the ranges of the two input `Interval` objects do not overlap, the output is an array of `Interval` objects covering the union of the ranges of the inputs.

```
interval3 = fixed.Interval(100, 200)
```

```
interval3 = 
    [100,200]

  1x1 fixed.Interval with properties:

             LeftEnd: 100
            RightEnd: 200
        IsLeftClosed: true
       IsRightClosed: true
```

```
intervalUnion = union(interval1, interval3)
```

```
intervalUnion = 
    [-10,10]    [100,200]

  1x2 fixed.Interval with properties:

             LeftEnd
            RightEnd
        IsLeftClosed
       IsRightClosed
```

Input Arguments

`A, B` — Input `fixed.Interval` objects
`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` objects, specified as `fixed.Interval` objects, or arrays of `fixed.Interval` objects.

Output Arguments

`C` — Union of `fixed.Interval` objects
`fixed.Interval` object | array of `fixed.Interval` objects

Union of input `fixed.Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

The output `Interval` object contains all values in `A` or `B`.

Version History
Introduced in R2019b

See Also

`fixed.Interval` | `contains` | `intersect` | `overlaps` | `setdiff`

unique

Package: `fixed`

Get set of unique values in `fixed.Interval` object

Syntax

`uniqueinterval = unique(interval)`

Description

`uniqueinterval = unique(interval)` returns a vector of incrementally sorted and non overlapping intervals that represent an equivalent value set as `fixed.Interval` object, `interval`.

Examples

Create a non-overlapping set of intervals from an array of `Interval` objects

Use the `unique` function to get a non-overlapping set of intervals from an array of `Interval` objects.

```
intervals = fixed.Interval({-5,5},{-10,10},{4,20},{50,100})
```

```
    [-5,5]    [-10,10]    [4,20]    [50,100]

  1x4 fixed.Interval with properties:

             LeftEnd
            RightEnd
        IsLeftClosed
       IsRightClosed
```

The first three intervals represented in the object overlap with one another. The fourth interval is disjointed from the set.

```
uniqueInterval = unique(intervals)
```

```
uniqueInterval = 

    [-10,20]    [50,100]

  1x2 fixed.Interval with properties:

             LeftEnd
            RightEnd
        IsLeftClosed
       IsRightClosed
```

The output, `uniqueInterval`, an array of two `Interval` objects, merges the three overlapping intervals into a single `Interval` object.

Input Arguments

`interval` — `fixed.Interval` object
`fixed.Interval` object | array of `fixed.Interval` objects

Input `fixed.Interval` object, specified as a `fixed.Interval` object, or an array of `fixed.Interval` objects.

Output Arguments

`uniqueinterval` — Non-overlapping set of `Interval` objects
`fixed.Interval` object | array of `fixed.Interval` objects

Non-overlapping set of `Interval` objects, returned as a `fixed.Interval` object or an array of `fixed.Interval` objects.

When `interval` is a scalar `Interval` object, the output is the same as the input.

Version History

Introduced in R2019b

See Also

`fixed.Interval` | `contains` | `intersect` | `overlaps` | `setdiff` | `union`

quantize

Quantize `fi` values using `fixed.Quantizer` object

Note `quantize` and `fixed.Quantizer` are not recommended. Use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. For more information, see Compatibility Considerations.

Syntax

```
y = quantize(q,x)
```

Description

`y = quantize(q,x)` uses the `fixed.Quantizer` object `q` to quantize `x`. `x` can be any fixed-point `fi` number except a Boolean value.

- If `x` is a scaled double, the data of the output `y` will be the same as the data of the input `x`. Only the fixed-point settings of `y` will change.
- When `x` is a double or single, then `y = x`. This functionality allows you to share the same code for both floating-point data types and fixed-point `fi` data types when quantizers are present.

Examples

Reduce Word Length Resulting From Adding Two Fixed-Point Numbers

Use `fixed.Quantizer` to reduce the word length that results from adding two fixed-point numbers.

```
q = fixed.Quantizer
x1 = fi(0.1,1,16,15);
x2 = fi(0.8,1,16,15);
y  = quantize(q,x1+x2)
```

```
q = 

  fixed.Quantizer with properties:

                   Signed: 1
               WordLength: 16
    SlopeAdjustmentFactor: 1
           FixedExponent: -15
                     Bias: 0
               Signedness: 'Signed'
                    Slope: 3.0518e-05
          FractionLength: 15
          RoundingMethod: 'Floor'
          OverflowAction: 'Wrap'


y = 
```

```
    0.9000

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

Quantize Binary-Point Scaled Fixed-Point fi to Slope-Bias Scaled Fixed-Point fi

Use a `fixed.Quantizer` object to change a binary-point scaled fixed-point `fi` to a slope-bias scaled fixed-point `fi`.

```
x = fi(pi,1,16,13)
q = fixed.Quantizer(numerictype(1,7,1.6,0.2),'Round','Saturate')
y = quantize(q,x)
```

```
x =

    3.1416

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 13

q =

  fixed.Quantizer with properties:

                  Signed: 1
              WordLength: 7
    SlopeAdjustmentFactor: 1.6000
            FixedExponent: 0
                     Bias: 0.2000
               Signedness: 'Signed'
                    Slope: 1.6000
           FractionLength: 0
           RoundingMethod: 'Round'
           OverflowAction: 'Saturate'


y =

    3.4000

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 7
                 Slope: 1.6
                  Bias: 0.2
```

Input Arguments

q — Data type properties
`fixed.Quantizer` object

Data type properties to use for quantization, specified as a `fixed.Quantizer` object.

x — Data to quantize
`fi` object

Data to quantize, specified as a `fi` object.

Data Types: `fi`

Version History

Introduced in R2011b

R2013a: `quantize` is not recommended
Not recommended starting in R2013a

`quantize` and `fixed.Quantizer` are not recommended. Use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. There are no plans to remove `fixed.Quantizer`.

Starting in R2013a, use `cast`, `zeros`, `ones`, `eye`, or `subsasgn` instead. The `cast`, `zeros`, `ones`, `eye`, and `subsasgn` functions can quantize other data types in addition to `fi` objects.

<table>
<tr><th>Not Recommended</th><th>Recommended</th></tr>
<tr><td>

```
x = fi(pi,1,16,13);

q = fixed.Quantizer(numerictype(1,7,1.6,0.2),'Round','Saturate');
y = quantize(q,x)

y = 

    3.4000

          DataTypeMode: Fixed-point: slope and bias scaling
            Signedness: Signed
            WordLength: 7
                 Slope: 1.6
                  Bias: 0.2
```

</td><td>

```
x = fi(pi,1,16,13);

F = fimath('RoundingMethod','Round','OverflowAction','Sa
nt = fi([],1,7,1.6,0.2,F);
y = cast(x,'like',nt)

y = 

    3.4000

          DataTypeMode: Fixed-point: slope and bias sca
            Signedness: Signed
            WordLength: 7
                 Slope: 1.6
                  Bias: 0.2
```

</td></tr>
</table>

See Also

`fixed.Quantizer`

FunctionApproximation.compressLookupTables

Compress all Lookup Table blocks in a system

Syntax

```
CompressionResult = FunctionApproximation.compressLookupTables(system)
CompressionResult = FunctionApproximation.compressLookupTables(system,
Name,Value)
```

Description

`CompressionResult = FunctionApproximation.compressLookupTables(system)` compresses all n-D Lookup Table blocks in the specified system. The compressed Lookup Table blocks output the same numerical results as the original Lookup Table blocks within the bounds of the breakpoints.

You can achieve additional memory savings by compressing each lookup table in the model individually and specifying tolerances for the compressed lookup table.

`CompressionResult = FunctionApproximation.compressLookupTables(system, Name,Value)` compresses all n-D Lookup Table blocks in the specified system with additional properties specified by name and value pair arguments.

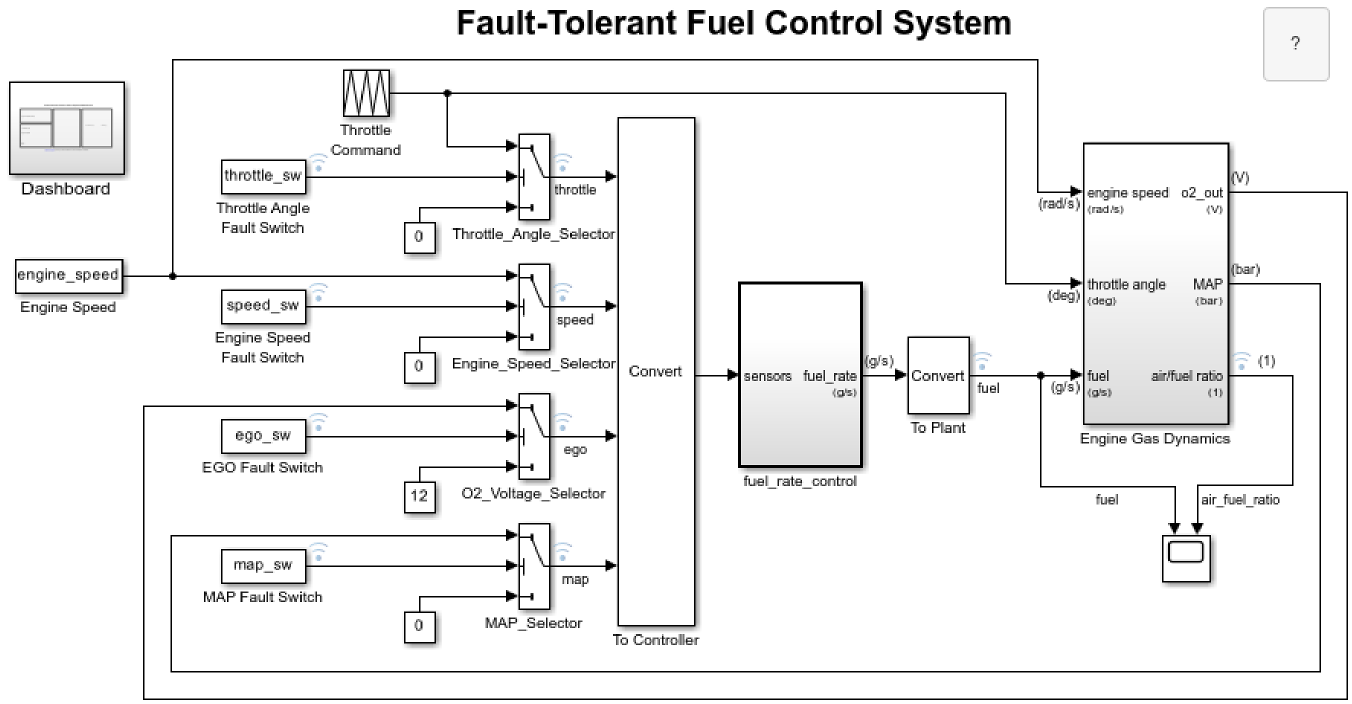
Examples

Compress All Lookup Table Blocks in a System

This example shows how to compress all Lookup Table blocks in a system.

Open the model containing the lookup tables that you want to compress.

```
system = 'sldemo_fuelsys';
open_system(system)
```



Open the Dashboard subsystem to simulate any combination of sensor failures.

Use the `FunctionApproximation.compressLookupTables` function to compress all of the lookup tables in the model. The output specifies all blocks that are modified and the memory savings for each.

```
compressionResult = FunctionApproximation.compressLookupTables(system)
```

```
- Found 5 supported lookup tables
- Percent reduction in memory for compressed solution
    -  2.37% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant
    -  2.37% for sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throt
    -  3.55% for sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estima
    -  6.38% for sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Es
    -  9.38% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki

compressionResult = 

  LUTCompressionResult with properties:

                MemoryUnits: bytes
           MemoryUsageTable: [5x5 table]
               NumLUTsFound: 5
            NumImprovements: 5
            TotalMemoryUsed: 6024
         TotalMemoryUsedNew: 5796
         TotalMemorySavings: 228
    TotalMemorySavingsPercent: 3.7849
                        SUD: 'sldemo_fuelsys'
```

```
WordLengths: [8 16 32]
FindOptions: [1x1 Simulink.internal.FindOptions]
    Display: 1
```

Use the `replace` function to replace each Lookup Table block with a block containing the original and compressed version of the lookup table.

```
replace(compressionResult);
```

You can revert the lookup tables back to their original state using the `revert` function.

```
revert(compressionResult);
```

Input Arguments

`system` — Name of model or subsystem in which to compress all Lookup Table blocks
character vector

Name of model or subsystem in which to compress all n-D Lookup Table blocks, specified as a character vector.

Example: `compressionResult = FunctionApproximation.compressLookupTables('sldemo_fuelsys');`

Data Types: `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `CompressionResult = FunctionApproximation.compressLookupTables('sldemo_fuelsys', 'WordLengths', [8,16,32])`

`Display` — Whether to display details of each iteration of the optimization
`true` (default) | `false`

Whether to display details of each iteration of the optimization, specified as a logical. A value of `1` results in information in the command window at each iteration of the approximation process. A value of `0` does not display information until the approximation is complete.

Data Types: `logical`

`WordLengths` — Word lengths permitted in the lookup table approximate
integer scalar | integer vector

Specify the word lengths, in bits, that can be used in the lookup table approximate based on your intended hardware. For example, if you intend to target an embedded processor, you can restrict the data types in your lookup table to native types, 8, 16, and 32. The word lengths must be between 1 and 128.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `fi`

`FindOptions` — Options for finding lookup tables in system
`Simulink.FindOptions` object

`Simulink.FindOptions` object specifying options for finding lookup tables in the system.

Output Arguments

`CompressionResult` — `LUTCompressionResult` object created during compression of lookup tables
`LUTCompressionResult` object

Compression result object created during compression of the Lookup Table blocks in the model, returned as a `LUTCompressionResult` object.

Version History

Introduced in R2020a

See Also

Classes
`LUTCompressionResult`

Functions
`replace` | `revert`

lutmemoryusage

Class: FunctionApproximation.LUTMemoryUsageCalculator
Package: FunctionApproximation

Calculate memory used by lookup table blocks in a system

Syntax

```
memory = lutmemoryusage(calculator,system)
```

Description

`memory = lutmemoryusage(calculator,system)` calculates the memory used by each lookup table block in the specified model or subsystem.

Input Arguments

`calculator` — `FunctionApproximation.LUTMemoryUsageCalculator` object
`FunctionApproximation.LUTMemoryUsageCalculator`

`FunctionApproximation.LUTMemoryUsageCalculator` object.

`system` — Model or subsystem containing lookup table blocks
character vector

Model or subsystem containing lookup table blocks, specified as a character vector.

Data Types: `char`

Output Arguments

`memory` — Memory used by the lookup tables in the system
table

Table displaying the memory, in bits, used by each lookup table block in the specified system.

Examples

Calculate the Total Memory Used by Lookup Tables in a Model

Use the `FunctionApproximation.LUTMemoryUsageCalculator` class to calculate the memory used by lookup table blocks in a model.

Create a `FunctionApproximation.LUTMemoryUsageCalculator` object.

```
calculator = FunctionApproximation.LUTMemoryUsageCalculator
```

Use the `lutmemoryusage` method to get the memory used by each lookup table block in the `sldemo_fuelsys` model.

```
openExample('simulink_automotive/ModelingAFaultTolerantFuelControlSystemExample',...
    'supportingfile','sldemo_fuelsys');
lutmemoryusage(calculator,'sldemo_fuelsys')

ans =

  5×2 table

                                            BlockPath
         _________________________________________________________________________________________

    1    "sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant"
    2    "sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throttle Esti
    3    "sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estimation"
    4    "sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Estimatio
    5    "sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki"
```

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTMemoryUsageCalculator` | `FunctionApproximation.LUTSolution`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

approximate

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Generate a Lookup Table block or lookup table as a MATLAB function from a FunctionApproximation.LUTSolution

Syntax

```
approximate(solution)
approximate(solution,'Name',fileName)
approximate(solution,'Name',fileName,'Path',filePath)
```

Description

`approximate(solution)` generates either a Simulink model containing a subsystem made up of the Lookup Table block, or a lookup table as a MATLAB function, depending on the `ApproximateSolutionType` property of the `FunctionApproximation.Options` object. Data and breakpoints of the generated lookup table are specified by the `FunctionApproximation.LUTSolution` object, `solution`. The generated Lookup Table block is surrounded with Data Type Conversion blocks.

`approximate(solution,'Name',fileName)` generates a lookup table as a MATLAB function with the name of the generated .m script specified by `fileName`. This option is only available when the `ApproximateSolutionType` property of `FunctionApproximation.Options` is set to `MATLAB`.

`approximate(solution,'Name',fileName,'Path',filePath)` generates a lookup table as a MATLAB function with the file path for the generated .m script specified by `filePath`. This option is only available when the `ApproximateSolutionType` property of `FunctionApproximation.Options` is set to `MATLAB`.

Input Arguments

solution — Solution to generate lookup table from
`FunctionApproximation.LUTSolution` object

The solution to generate a lookup table from, specified as a `FunctionApproximation.LUTSolution` object.

fileName — File name for generated MATLAB function
`approximateFunction_timeStamp` (default) | character array

File name for generated MATLAB function, specified as a character array. If no custom file name is specified, a time stamp is used to generate a unique file name. For example, `approximateFunction_20210617T111033122.m`.

Example: `approximate(solution,'Name','myLUT')`

Data Types: `char`

filePath — File path for generated MATLAB function
current working directory (default) | character array

File path for generated MATLAB function, specified as a character array. If no custom file path name is specified, the current working directory is used.

Example: `approximate(solution,'Name','myLUT','Path','C:\Users\myPath')`

Data Types: `char`

Examples

Generate a Lookup Table Approximating a Function

Create a `FunctionApproximation.Problem` object defining the function you want to approximate.

```
problem = FunctionApproximation.Problem('tanh')
```

```
problem = 

  1x1 FunctionApproximation.Problem with properties:

    FunctionToApproximate: @(x)tanh(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(1,16,12)"
         InputLowerBounds: -8
         InputUpperBounds: 8
               OutputType: "numerictype(1,16,15)"
                  Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. Approximate the `tanh` function using the `solve` method.

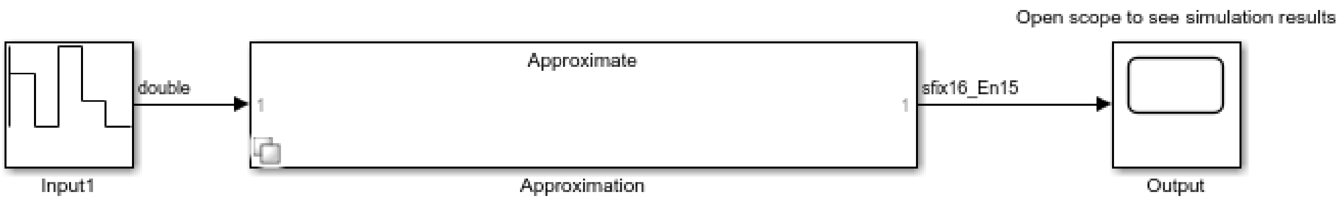
```
solution = solve(problem)
```

```
Searching for fixed-point solutions.

|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   0 |             64 |        0 |          2 |              16 |           16 |             Eve
|   1 |           1248 |        1 |         76 |              16 |           16 |             Eve
|   2 |           1232 |        1 |         75 |              16 |           16 |             Eve
|   3 |            944 |        1 |         57 |              16 |           16 |             Eve
|   4 |            928 |        1 |         56 |              16 |           16 |             Eve
|   5 |            656 |        0 |         39 |              16 |           16 |             Eve
|   6 |            640 |        0 |         38 |              16 |           16 |             Eve
|   7 |            784 |        1 |         47 |              16 |           16 |             Eve
|   8 |            704 |        1 |         42 |              16 |           16 |             Eve
|   9 |            672 |        1 |         40 |              16 |           16 |             Eve
|  10 |            368 |        0 |         21 |              16 |           16 |             Eve
|  11 |            512 |        0 |         30 |              16 |           16 |             Eve
|  12 |            592 |        0 |         35 |              16 |           16 |             Eve
|  13 |            624 |        0 |         37 |              16 |           16 |             Eve
|  14 |            384 |        1 |         12 |              16 |           16 |         Explic
|  15 |            384 |        0 |         12 |              16 |           16 |         Explic
|  16 |            384 |        1 |         12 |              16 |           16 |         Explic
```

```
Best Solution
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  14 |            384 |        1 |         12 |              16 |           16 |          Explic


solution = 

  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 14
    Feasible: "true"
```

Generate a Simulink™ subsystem containing a Lookup Table block approximating the `tanh` function.

```
approximate(solution)
```



Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTSolution` | `FunctionApproximation.LUTMemoryUsageCalculator`

Functions
`solve` | `compare`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"
"Generate an Optimized Lookup Table as a MATLAB Function Programmatically"
"Generate an Optimized Lookup Table as a MATLAB Function"

compare

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Compare numerical results of FunctionApproximation.LUTSolution to original function or lookup table

Syntax

data = compare(solution)

Description

data = compare(solution) plots the difference between the data contained in the FunctionApproximation.LUTSolution object, solution, and the original lookup table, function, or Math Function block.

Input Arguments

solution — Solution to compare original behavior against
FunctionApproximation.LUTSolution object

The solution to compare original behavior against, specified as a FunctionApproximation.LUTSolution object.

Output Arguments

data — Struct containing data comparing original and the solution
struct

Struct containing data comparing the original function or lookup table and the approximation contained in the solution.

Examples

Compare Function Approximation to Original Function

Create a FunctionApproximation.Problem object defining the function you want to approximate.

```
problem = FunctionApproximation.Problem('tanh')
```

```
problem = 
  1x1 FunctionApproximation.Problem with properties:

    FunctionToApproximate: @(x)tanh(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(1,16,12)"
         InputLowerBounds: -8
         InputUpperBounds: 8
               OutputType: "numerictype(1,16,15)"
```

```
                    Options: [1x1 FunctionApproximation.Options]
```

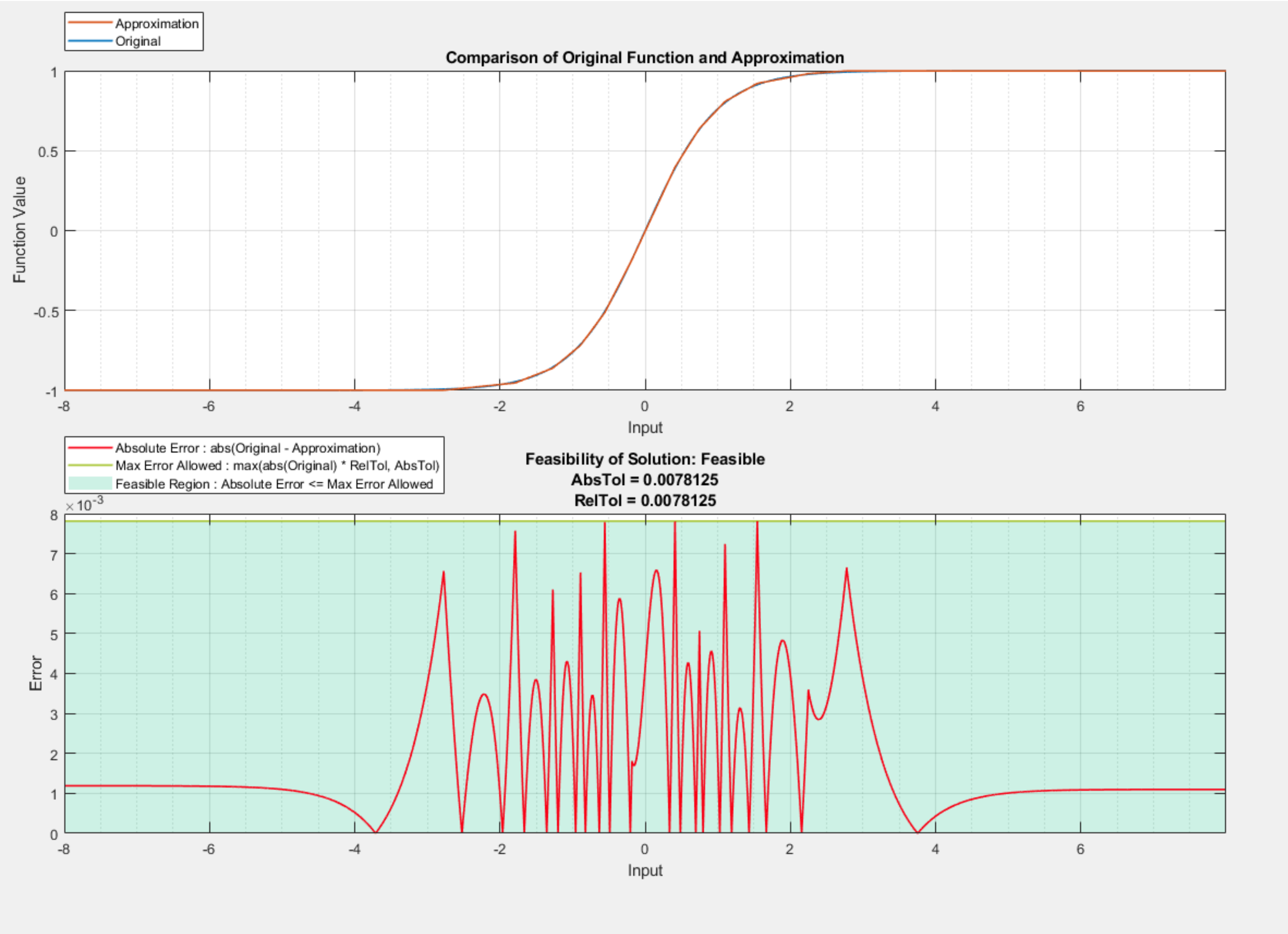
Use default values for all other options. Approximate the `tanh` function using the `solve` method.

```
solution = solve(problem)
```

```
Searching for fixed-point solutions.

|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   0 |             64 |        0 |          2 |              16 |           16 |             Eve
|   1 |           1248 |        1 |         76 |              16 |           16 |             Eve
|   2 |           1232 |        1 |         75 |              16 |           16 |             Eve
|   3 |            944 |        1 |         57 |              16 |           16 |             Eve
|   4 |            928 |        1 |         56 |              16 |           16 |             Eve
|   5 |            656 |        0 |         39 |              16 |           16 |             Eve
|   6 |            640 |        0 |         38 |              16 |           16 |             Eve
|   7 |            784 |        1 |         47 |              16 |           16 |             Eve
|   8 |            704 |        1 |         42 |              16 |           16 |             Eve
|   9 |            672 |        1 |         40 |              16 |           16 |             Eve
|  10 |            368 |        0 |         21 |              16 |           16 |             Eve
|  11 |            512 |        0 |         30 |              16 |           16 |             Eve
|  12 |            592 |        0 |         35 |              16 |           16 |             Eve
|  13 |            624 |        0 |         37 |              16 |           16 |             Eve
|  14 |            384 |        1 |         12 |              16 |           16 |          Explic
|  15 |            384 |        0 |         12 |              16 |           16 |          Explic
|  16 |            384 |        1 |         12 |              16 |           16 |          Explic

Best Solution
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  14 |            384 |        1 |         12 |              16 |           16 |          Explic

solution = 
  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 14
    Feasible: "true"
```

Compare the original function and the function approximation.

```
data = compare(solution)
```




```
data = struct with fields:
    Breakpoints: [65536x1 double]
       Original: [65536x1 double]
    Approximate: [65536x1 double]
```

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTSolution` | `FunctionApproximation.LUTMemoryUsageCalculator`

Functions
`solve` | `approximate`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

displayallsolutions

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Display all solutions found during function approximation

Syntax

```
displayallsolutions(solution)
```

Description

`displayallsolutions(solution)` displays all solutions, including the non-feasible solutions, associated with a `FunctionApproximation.LUTSolution` object.

Input Arguments

solution — Solution object from which to display all associated solutions
`FunctionApproximation.LUTSolution` object

`FunctionApproximation.LUTSolution` object from which to display all associated solutions.

Examples

Display All Solutions Found During Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Display all solutions found during the approximation process using the `displayallsolutions` method.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x)sin(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(0,16,13)"
         InputLowerBounds: 0
         InputUpperBounds: 6.2832
               OutputType: "numerictype(1,16,14)"
                  Options: [1×1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
solution = 

  FunctionApproximation.LUTSolution with properties
```

```
          ID: 8
    Feasible: "true"
```

```
displayallsolutions(solution)
```

```
|    ID |   Memory (bits) |   ConstraintMet |      Table Size | Breakpoints WLs |    TableData WL
|     0 |              64 |               0 |               2 |              16 |              16
|     1 |             464 |               0 |              27 |              16 |              16
|     2 |             864 |               1 |              52 |              16 |              16
|     3 |              64 |               0 |               2 |              16 |              16
|     4 |             560 |               1 |              33 |              16 |              16
|     5 |             304 |               0 |              17 |              16 |              16
|     6 |             432 |               0 |              25 |              16 |              16
|     7 |             496 |               1 |              29 |              16 |              16
|     8 |             464 |               1 |              27 |              16 |              16
|     9 |             448 |               0 |              26 |              16 |              16
|    10 |             704 |               1 |              22 |              16 |              16

Best Solution
|    ID |   Memory (bits) |   ConstraintMet |      Table Size | Breakpoints WLs |    TableData WL
|     8 |             464 |               1 |              27 |              16 |              16
```

Version History

Introduced in R2018a

See Also

Apps

Lookup Table Optimizer

Classes

`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTMemoryUsageCalculator` | `FunctionApproximation.LUTSolution`

Functions

`totalmemoryusage` | `solutionfromID` | `displayfeasiblesolutions`

Topics

"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

displayfeasiblesolutions

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Display all feasible solutions found during function approximation

Syntax

```
displayfeasiblesolutions(solution)
```

Description

`displayfeasiblesolutions(solution)` displays all feasible solutions found during the approximation process, including the best solution. Feasible solutions are defined as any solutions to the original `FunctionApproximation.Problem` object that met the constraints defined in the associated `FunctionApproximation.Options` object.

Input Arguments

`solution` — Solution object from which to display all associated feasible solutions
`FunctionApproximation.LUTSolution` object

`FunctionApproximation.LUTSolution` object from which to display all associated feasible solutions.

Examples

Display All Feasible Solutions Found During Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Display all feasible solutions found during the approximation process using the `displayfeasiblesolutions` method.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x)sin(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(0,16,13)"
         InputLowerBounds: 0
         InputUpperBounds: 6.2832
               OutputType: "numerictype(1,16,14)"
                  Options: [1×1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
solution = 

  FunctionApproximation.LUTSolution with properties

          ID: 8
    Feasible: "true"
```

```
displayfeasiblesolutions(solution)
```

```
|    ID |   Memory (bits) |   ConstraintMet |      Table Size | Breakpoints WLs |    TableData WL
|     2 |             864 |               1 |              52 |              16 |              16
|     4 |             560 |               1 |              33 |              16 |              16
|     7 |             496 |               1 |              29 |              16 |              16
|     8 |             464 |               1 |              27 |              16 |              16
|    10 |             704 |               1 |              22 |              16 |              16

Best Solution
|    ID |   Memory (bits) |   ConstraintMet |      Table Size | Breakpoints WLs |    TableData WL
|     8 |             464 |               1 |              27 |              16 |              16
```

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
FunctionApproximation.Problem | FunctionApproximation.Options | FunctionApproximation.LUTMemoryUsageCalculator | FunctionApproximation.LUTSolution

Functions
compare | totalmemoryusage | solutionfromID | displayallsolutions

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

getErrorValue

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Get the total error of the lookup table approximation

Syntax

`memory = getErrorValue(solution)`

Description

`memory = getErrorValue(solution)` returns the total error of the lookup table approximation specified by `solution`.

Input Arguments

`solution` — Solution to get error of
`FunctionApproximation.LUTSolution` object

Solution to get error of, specified as a `FunctionApproximation.LUTSolution` object.

Output Arguments

`error` — Total error of the lookup table approximation
struct

Total error of the lookup table approximation, returned as a struct.

The struct contains two fields. The `MaxErrorInSolution` field specifies the maximum difference between the original function or block and the lookup table approximation. The `ErrorUpperBound` field displays the maximum error that was acceptable according to the tolerances specified on the `FunctionApproximation.Options` object.

Examples

Calculate the Total Error of a Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Calculate the total error of the `FunctionApproximation.LUTSolution` object using the `getErrorValue` method.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem =
```

```
  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x)sin(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(0,16,13)"
         InputLowerBounds: 0
         InputUpperBounds: 6.2832
               OutputType: "numerictype(1,16,14)"
                  Options: [1×1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
solution = 

  FunctionApproximation.LUTSolution with properties

          ID: 8
    Feasible: "true"
```

```
error = getErrorValue(solution)
```

```
error = 

  struct with fields:

    MaxErrorInSolution: 0.0073
       ErrorUpperBound: 0.0078
```

Version History

Introduced in R2019a

See Also

`FunctionApproximation.LUTSolution`

Topics

"Approximate Functions with a Direct Lookup Table"
"Optimize Lookup Tables for Memory-Efficiency Programmatically"

replaceWithApproximate

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Replace block with the generated lookup table approximation

Syntax

`replaceWithApproximate(solution)`

Description

`replaceWithApproximate(solution)` replaces the simulink block with its lookup table approximation, generated using the `approximate` method of the `FunctionApproximation.LUTSolution` object.

Input Arguments

`solution` — Solution to use to replace the source block
`FunctionApproximation.LUTSolution` object

Solution to replace the source block, specified as a `FunctionApproximation.LUTSolution` object.

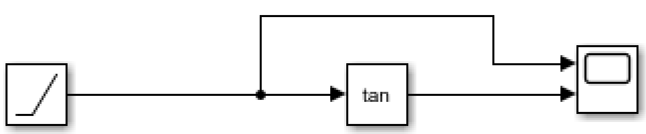
Examples

Replace a Block with an Approximation

This example shows how to approximate a block using a lookup table approximation, replace the original block with the approximation, and then revert the block back to its original state.

Open the model containing the block to approximate. In this example, replace the tan block with a lookup table approximation.

```
open_system('ex_luto_approx')
```



Create a `FunctionApproximation.Problem` object specifying what you want to approximate.

```
problem = FunctionApproximation.Problem('ex_luto_approx/Trigonometric Function')
```

```
problem = 

  1x1 FunctionApproximation.Problem with properties:
```

```
    FunctionToApproximate: 'ex_luto_approx/Trigonometric Function'
           NumberOfInputs: 1
               InputTypes: "numerictype('double')"
         InputLowerBounds: -1.5083
         InputUpperBounds: 1.5083
               OutputType: "numerictype('double')"
                  Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. To approximate the block use the `solve` method.

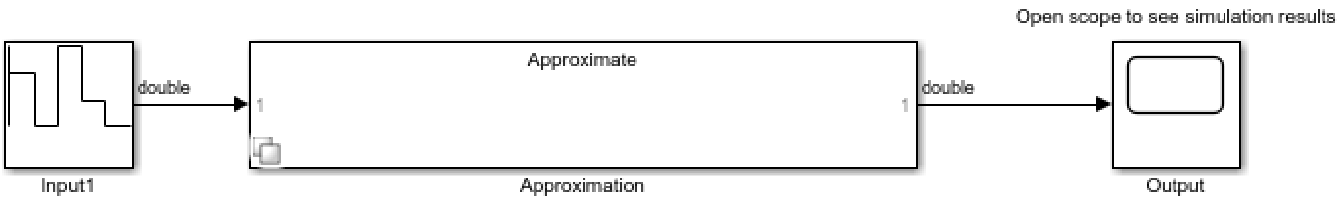
```
solution = solve(problem)
```

```
Searching for fixed-point solutions.

|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   0 |             48 |        0 |          2 |               8 |           16 |             Eve
|   1 |            800 |        0 |         49 |               8 |           16 |             Eve
|   2 |           1584 |        1 |         98 |               8 |           16 |             Eve
|   3 |           1056 |        0 |         65 |               8 |           16 |             Eve
|   4 |            544 |        0 |         33 |               8 |           16 |             Eve
|   5 |            416 |        0 |         25 |               8 |           16 |             Eve
|   6 |            368 |        0 |         22 |               8 |           16 |             Eve
|   7 |             64 |        0 |          2 |              16 |           16 |             Eve
|   8 |            768 |        1 |         46 |              16 |           16 |             Eve
|   9 |            752 |        1 |         45 |              16 |           16 |             Eve
|  10 |            592 |        1 |         35 |              16 |           16 |             Eve
|  11 |            576 |        1 |         34 |              16 |           16 |             Eve
|  12 |            416 |        0 |         24 |              16 |           16 |             Eve
|  13 |            400 |        0 |         23 |              16 |           16 |             Eve
|  14 |            496 |        0 |         29 |              16 |           16 |             Eve
|  15 |            528 |        1 |         31 |              16 |           16 |             Eve
|  16 |            512 |        0 |         30 |              16 |           16 |             Eve
|  17 |            288 |        0 |         16 |              16 |           16 |             Eve
|  18 |            464 |        0 |         27 |              16 |           16 |             Eve
|  19 |             80 |        0 |          2 |               8 |           32 |             Eve
|  20 |             48 |        0 |          2 |               8 |           16 |         EvenPow
|  21 |            416 |        0 |         25 |               8 |           16 |         EvenPow
|  22 |            224 |        0 |         13 |               8 |           16 |         EvenPow
|  23 |             64 |        0 |          2 |              16 |           16 |         EvenPow
|  24 |            432 |        0 |         25 |              16 |           16 |         EvenPow
|  25 |            240 |        0 |         13 |              16 |           16 |         EvenPow
|  26 |             80 |        0 |          2 |               8 |           32 |         EvenPow
|  27 |            432 |        0 |         13 |               8 |           32 |         EvenPow
|  28 |             96 |        0 |          2 |              16 |           32 |         EvenPow
|  29 |            448 |        0 |         13 |              16 |           32 |         EvenPow
|  30 |            128 |        0 |          2 |              32 |           32 |         EvenPow
|  31 |            480 |        0 |         13 |              32 |           32 |         EvenPow
|  32 |             96 |        0 |          2 |              32 |           16 |         EvenPow
|  33 |            464 |        0 |         25 |              32 |           16 |         EvenPow
|  34 |            272 |        0 |         13 |              32 |           16 |         EvenPow
|  35 |            216 |        1 |          9 |               8 |           16 |          Explic
|  36 |            192 |        0 |          8 |               8 |           16 |          Explic
|  37 |            192 |        0 |          8 |               8 |           16 |          Explic
|  38 |            192 |        0 |          8 |               8 |           16 |          Explic
|  39 |            192 |        0 |          8 |               8 |           16 |          Explic
|  40 |            192 |        1 |          8 |               8 |           16 |          Explic
```

```
Searching for floating-point solutions.

|  41 |              64 |        0 |          2 |                16 |            16 |              Eve
|  42 |             768 |        0 |         46 |                16 |            16 |              Eve
|  43 |             752 |        1 |         45 |                16 |            16 |              Eve
|  44 |             160 |        0 |          2 |                16 |            64 |              Eve
|  45 |              64 |        0 |          2 |                16 |            16 |          EvenPow
|  46 |             160 |        0 |          2 |                16 |            64 |          EvenPow

Best Solution
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  40 |             192 |        1 |          8 |                 8 |            16 |            Explic


solution = 

  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 40
    Feasible: "true"
```

Generate a Simulink™ subsystem containing the lookup table approximation using the `approximate` method.

```
approximate(solution)
```



Replace the original block with the approximation.

```
replaceWithApproximate(solution)
```

You can revert the system back to its original state using the `revertToOriginal` method.

```
revertToOriginal(solution)
```

Version History

Introduced in R2018b

See Also

`revertToOriginal` | `approximate`

Topics

"Approximate Functions with a Direct Lookup Table"
"Optimize Lookup Tables for Memory-Efficiency Programmatically"

revertToOriginal

Class: `FunctionApproximation.LUTSolution`
Package: `FunctionApproximation`

Revert the block that was replaced by the approximation back to its original state

Syntax

```
revertToOriginal(solution)
```

Description

`revertToOriginal(solution)` reverts the block that was replaced by a lookup table approximation back to its original state.

Note You can only revert a block back to its original state within a single MATLAB session.

Input Arguments

`solution` — Solution approximating the block you want to revert to its original state
`FunctionApproximation.LUTSolution` object

The solution approximating the block you want to revert to its original state, specified as a `FunctionApproximation.LUTSolution` object.

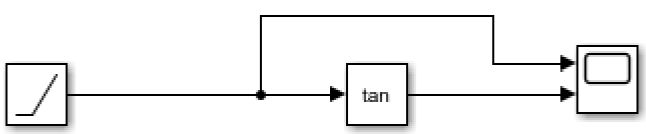
Examples

Replace a Block with an Approximation

This example shows how to approximate a block using a lookup table approximation, replace the original block with the approximation, and then revert the block back to its original state.

Open the model containing the block to approximate. In this example, replace the tan block with a lookup table approximation.

```
open_system('ex_luto_approx')
```



Create a `FunctionApproximation.Problem` object specifying what you want to approximate.

```
problem = FunctionApproximation.Problem('ex_luto_approx/Trigonometric Function')
```

```
problem =
```

```
  1x1 FunctionApproximation.Problem with properties:

    FunctionToApproximate: 'ex_luto_approx/Trigonometric Function'
           NumberOfInputs: 1
               InputTypes: "numerictype('double')"
         InputLowerBounds: -1.5083
         InputUpperBounds: 1.5083
               OutputType: "numerictype('double')"
                  Options: [1x1 FunctionApproximation.Options]
```

Use default values for all other options. To approximate the block use the `solve` method.

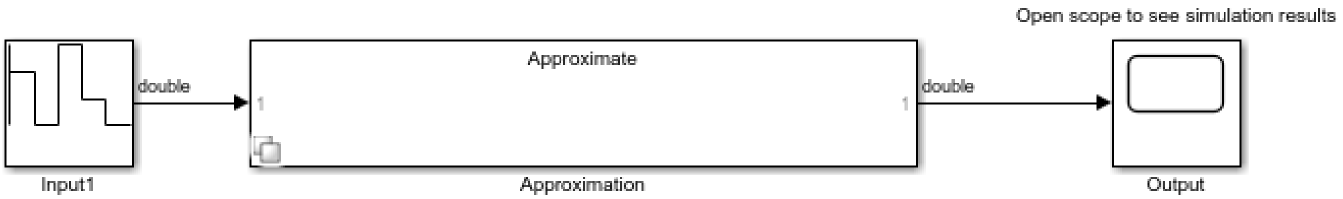
```
solution = solve(problem)
```

```
Searching for fixed-point solutions.

|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   0 |             48 |        0 |          2 |               8 |           16 |            Eve
|   1 |            800 |        0 |         49 |               8 |           16 |            Eve
|   2 |           1584 |        1 |         98 |               8 |           16 |            Eve
|   3 |           1056 |        0 |         65 |               8 |           16 |            Eve
|   4 |            544 |        0 |         33 |               8 |           16 |            Eve
|   5 |            416 |        0 |         25 |               8 |           16 |            Eve
|   6 |            368 |        0 |         22 |               8 |           16 |            Eve
|   7 |             64 |        0 |          2 |              16 |           16 |            Eve
|   8 |            768 |        1 |         46 |              16 |           16 |            Eve
|   9 |            752 |        1 |         45 |              16 |           16 |            Eve
|  10 |            592 |        1 |         35 |              16 |           16 |            Eve
|  11 |            576 |        1 |         34 |              16 |           16 |            Eve
|  12 |            416 |        0 |         24 |              16 |           16 |            Eve
|  13 |            400 |        0 |         23 |              16 |           16 |            Eve
|  14 |            496 |        0 |         29 |              16 |           16 |            Eve
|  15 |            528 |        1 |         31 |              16 |           16 |            Eve
|  16 |            512 |        0 |         30 |              16 |           16 |            Eve
|  17 |            288 |        0 |         16 |              16 |           16 |            Eve
|  18 |            464 |        0 |         27 |              16 |           16 |            Eve
|  19 |             80 |        0 |          2 |               8 |           32 |            Eve
|  20 |             48 |        0 |          2 |               8 |           16 |        EvenPow
|  21 |            416 |        0 |         25 |               8 |           16 |        EvenPow
|  22 |            224 |        0 |         13 |               8 |           16 |        EvenPow
|  23 |             64 |        0 |          2 |              16 |           16 |        EvenPow
|  24 |            432 |        0 |         25 |              16 |           16 |        EvenPow
|  25 |            240 |        0 |         13 |              16 |           16 |        EvenPow
|  26 |             80 |        0 |          2 |               8 |           32 |        EvenPow
|  27 |            432 |        0 |         13 |               8 |           32 |        EvenPow
|  28 |             96 |        0 |          2 |              16 |           32 |        EvenPow
|  29 |            448 |        0 |         13 |              16 |           32 |        EvenPow
|  30 |            128 |        0 |          2 |              32 |           32 |        EvenPow
|  31 |            480 |        0 |         13 |              32 |           32 |        EvenPow
|  32 |             96 |        0 |          2 |              32 |           16 |        EvenPow
|  33 |            464 |        0 |         25 |              32 |           16 |        EvenPow
|  34 |            272 |        0 |         13 |              32 |           16 |        EvenPow
|  35 |            216 |        1 |          9 |               8 |           16 |          Explic
|  36 |            192 |        0 |          8 |               8 |           16 |          Explic
|  37 |            192 |        0 |          8 |               8 |           16 |          Explic
|  38 |            192 |        0 |          8 |               8 |           16 |          Explic
```

```
|  39 |             192 |        0 |          8 |                8 |           16 |          Explic
|  40 |             192 |        1 |          8 |                8 |           16 |          Explic
Searching for floating-point solutions.

|  41 |              64 |        0 |          2 |               16 |           16 |             Eve
|  42 |             768 |        0 |         46 |               16 |           16 |             Eve
|  43 |             752 |        1 |         45 |               16 |           16 |             Eve
|  44 |             160 |        0 |          2 |               16 |           64 |             Eve
|  45 |              64 |        0 |          2 |               16 |           16 |         EvenPow
|  46 |             160 |        0 |          2 |               16 |           64 |         EvenPow

Best Solution
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  40 |             192 |        1 |          8 |                8 |           16 |          Explic


solution = 

  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 40
    Feasible: "true"
```

Generate a Simulink™ subsystem containing the lookup table approximation using the `approximate` method.

```
approximate(solution)
```



Replace the original block with the approximation.

```
replaceWithApproximate(solution)
```

You can revert the system back to its original state using the `revertToOriginal` method.

```
revertToOriginal(solution)
```

Version History

Introduced in R2018b

See Also

`approximate` | `replaceWithApproximate`

Topics

"Approximate Functions with a Direct Lookup Table"

“Optimize Lookup Tables for Memory-Efficiency Programmatically”

solutionfromID

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Access a solution found during the approximation process

Syntax

```
other_solution = solutionfromID(solution,id)
```

Description

other_solution = solutionfromID(solution,id) returns the solution associated with the FunctionApproximation.LUTSolution object, solution, with the ID specified by id.

Input Arguments

solution — Solution object
FunctionApproximation.LUTSolution object

The solution object containing the solution you want to explore, specified as a FunctionApproximation.LUTSolution object.

id — ID of the solution
scalar integer

ID of the solution that you want to explore, specified as a scalar integer.

Data Types: double

Output Arguments

other_solution — FunctionApproximation.LUTSolution specified by id
FunctionApproximation.LUTSolution object

FunctionApproximation.LUTSolution object associated with the specified ID.

Examples

Examine Infeasible Function Approximation Solution

This example shows how to use the solutionfromID method of the FunctionApproximation.LUTSolution object to examine other approximation solutions.

Create a FunctionApproximation.Problem object defining a math function to approximate. Then use the solve method to get a FunctionApproximation.LUTSolution object.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem = 
  1x1 FunctionApproximation.Problem with properties:
```

```
    FunctionToApproximate: @(x)sin(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(0,16,13)"
         InputLowerBounds: 0
         InputUpperBounds: 6.2832
               OutputType: "numerictype(1,16,14)"
                  Options: [1x1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
Searching for fixed-point solutions.

|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   0 |             64 |        0 |          2 |              16 |           16 |             Eve
|   1 |            784 |        1 |         47 |              16 |           16 |             Eve
|   2 |            768 |        1 |         46 |              16 |           16 |             Eve
|   3 |            608 |        1 |         36 |              16 |           16 |             Eve
|   4 |            592 |        1 |         35 |              16 |           16 |             Eve
|   5 |            416 |        1 |         24 |              16 |           16 |             Eve
|   6 |            400 |        1 |         23 |              16 |           16 |             Eve
|   7 |            224 |        0 |         12 |              16 |           16 |             Eve
|   8 |            304 |        0 |         17 |              16 |           16 |             Eve
|   9 |            352 |        1 |         20 |              16 |           16 |             Eve
|  10 |            320 |        0 |         18 |              16 |           16 |             Eve
|  11 |            336 |        1 |         19 |              16 |           16 |             Eve
|  12 |             64 |        0 |          2 |              16 |           16 |         EvenPow
|  13 |            576 |        1 |         18 |              16 |           16 |          Explic
|  14 |            512 |        0 |         16 |              16 |           16 |          Explic
|  15 |            576 |        1 |         18 |              16 |           16 |          Explic

Best Solution
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  11 |            336 |        1 |         19 |              16 |           16 |             Eve

solution = 
  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 11
    Feasible: "true"
```

Display all feasible solutions found during the approximation process.

```
displayfeasiblesolutions(solution)
```

```
|  ID |  Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|   1 |            784 |        1 |         47 |              16 |           16 |             Eve
|   2 |            768 |        1 |         46 |              16 |           16 |             Eve
|   3 |            608 |        1 |         36 |              16 |           16 |             Eve
|   4 |            592 |        1 |         35 |              16 |           16 |             Eve
|   5 |            416 |        1 |         24 |              16 |           16 |             Eve
|   6 |            400 |        1 |         23 |              16 |           16 |             Eve
|   9 |            352 |        1 |         20 |              16 |           16 |             Eve
|  11 |            336 |        1 |         19 |              16 |           16 |             Eve
|  13 |            576 |        1 |         18 |              16 |           16 |          Explic
|  15 |            576 |        1 |         18 |              16 |           16 |          Explic
```

```
Best Solution
|  ID |   Memory (bits) | Feasible | Table Size | Breakpoints WLs | TableData WL | BreakpointSpeci
|  11 |             336 |        1 |         19 |              16 |           16 |              Eve
```

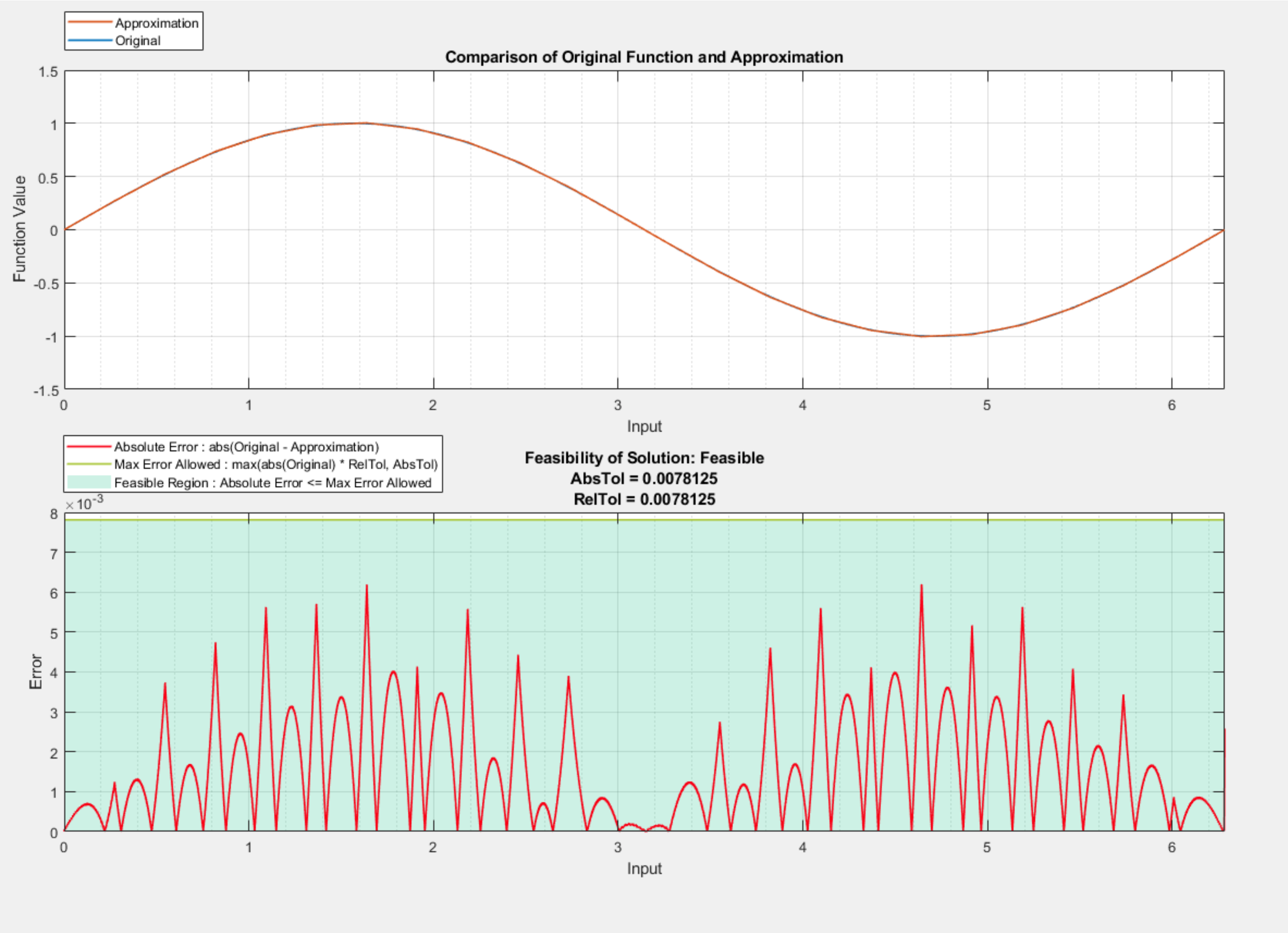
Solution with ID 5 is not listed as a feasible solution in the table. Explore this solution to see why it is not feasible.

```
solution5 = solutionfromID(solution, 5)
```

```
solution5 = 
  1x1 FunctionApproximation.LUTSolution with properties:

          ID: 5
    Feasible: "true"
```

Compare the numerical behavior of the solution with ID 5.

```
compare(solution5)
```




```
ans = struct with fields:
    Breakpoints: [51473x1 double]
       Original: [51473x1 double]
```

```
    Approximate: [51473x1 double]
```

You can see from the plot that the solution does not meet the required tolerances.

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTMemoryUsageCalculator` | `FunctionApproximation.LUTSolution`

Functions
`totalmemoryusage` | `displayfeasiblesolutions` | `displayallsolutions`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

totalmemoryusage

Class: FunctionApproximation.LUTSolution
Package: FunctionApproximation

Calculate total memory used by a lookup table approximation

Syntax

```
memory = totalmemoryusage(solution,units)
```

Description

`memory = totalmemoryusage(solution,units)` returns the total memory used by the lookup table approximation specified by `solution`, in the units specified by `units`.

Input Arguments

solution — Solution to get memory of
FunctionApproximation.LUTSolution object

Solution to get memory of, specified as a `FunctionApproximation.LUTSolution` object.

units — Units in which to display the total memory used
'bits' (default) | 'bytes' | 'GiB' | 'KiB' | 'MiB'

Units in which to display the total memory used, specified as a character vector.

Data Types: char

Output Arguments

memory — total memory used by a lookup table approximation
scalar

Total memory used by a lookup table approximation, returned as a scalar.

Examples

Calculate the Total Memory Used by a Lookup Table Approximation

Create a `FunctionApproximation.Problem` object defining a math function to approximate. Then, use the `solve` method to get a `FunctionApproximation.LUTSolution` object.

Calculate the total memory used by the `FunctionApproximation.LUTSolution` object using the `totalmemoryusage` method.

```
problem = FunctionApproximation.Problem('sin')
```

```
problem =
```

```
  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x)sin(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(0,16,13)"
         InputLowerBounds: 0
         InputUpperBounds: 6.2832
               OutputType: "numerictype(1,16,14)"
                  Options: [1×1 FunctionApproximation.Options]
```

```
solution = solve(problem)
```

```
solution = 

  FunctionApproximation.LUTSolution with properties

          ID: 8
    Feasible: "true"
```

```
totalmemoryusage(solution, 'bytes')
```

```
ans =

    58
```

Version History
Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTMemoryUsageCalculator` | `FunctionApproximation.LUTSolution`

Functions
`compare` | `solutionfromID` | `displayfeasiblesolutions` | `displayallsolutions`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

solve

Class: FunctionApproximation.Problem
Package: FunctionApproximation

Solve for optimized solution to function approximation problem

Syntax

```
solution = solve(problem)
```

Description

`solution = solve(problem)` solves the optimization problem defined by the `FunctionApproximation.Problem` object, `problem`, and returns the optimized result, `solution`, as a `FunctionApproximation.LUTSolution` object.

Input Arguments

problem — Optimization problem
FunctionApproximation.Problem

Optimization problem specified as a `FunctionApproximation.Problem` object defining the function or Math Function block to approximate, or the Lookup Table block to optimize, and other parameters and constraints to use during the optimization process.

Output Arguments

solution — Approximation solution
FunctionApproximation.LUTSolution object

Approximation solution, returned as a `FunctionApproximation.LUTSolution` object.

Examples

Approximate a Math Function

Create a `FunctionApproximation.Problem` object, specifying a math function to approximate.

```
problem = FunctionApproximation.Problem('log')
```

```
problem = 

  FunctionApproximation.Problem with properties

    FunctionToApproximate: @(x)log(x)
           NumberOfInputs: 1
               InputTypes: "numerictype(1,16,10)"
         InputLowerBounds: 0.6250
         InputUpperBounds: 15.6250
```

```
           OutputType: "numerictype(1,16,13)"
              Options: [1×1 FunctionApproximation.Options]
```

Use default values for all other options.

Use the `solve` method to generate an approximation of the function.

```
solution = solve(problem)
```

```
|    ID |   Memory (bits) |   ConstraintMet |      Table Size | Breakpoints WLs |    TableData WL
|     0 |              64 |               0 |               2 |              16 |              16
|     1 |            1984 |               1 |             122 |              16 |              16
|     2 |            1024 |               0 |              62 |              16 |              16
|     3 |            1968 |               1 |             121 |              16 |              16
|     4 |              64 |               0 |               2 |              16 |              16
|     5 |             416 |               1 |              13 |              16 |              16

Best Solution
|    ID |   Memory (bits) |   ConstraintMet |      Table Size | Breakpoints WLs |    TableData WL
|     5 |             416 |               1 |              13 |              16 |              16


solution = 

  FunctionApproximation.LUTSolution with properties

          ID: 5
    Feasible: "true"
```

You can then use the `approximate` method to generate a subsystem containing the lookup table approximation.

Version History

Introduced in R2018a

See Also

Apps
Lookup Table Optimizer

Classes
`FunctionApproximation.Problem` | `FunctionApproximation.Options` | `FunctionApproximation.LUTSolution` | `FunctionApproximation.LUTMemoryUsageCalculator`

Functions
`approximate` | `compare`

Topics
"Optimize Lookup Tables for Memory-Efficiency Programmatically"
"Optimize Lookup Tables for Memory-Efficiency"

addSpecification

Class: `fxpOptimizationOptions`

Specify known data types in a system

Syntax

```
addSpecification(options,Name,Value)
```

Description

`addSpecification(options,Name,Value)` specifies known data types in the model using name-value pairs. After specifying these known parameters, if you optimize the data types in a system, the optimization process will not change the specified block parameter data type. Specifications are applied to the model during evaluation and to the final model. Specifications are not considered during range collection.

You can use this method in cases where parts of a system are known to always be a certain data type. For example, if the input to your system comes from an 8-bit sensor.

Input Arguments

`options` — Associated `fxpOptimizationOptions` object
`fxpOptimizationOptions` object

`fxpOptimizationOptions` object in which to specify a known data type for a system.

Example: `opt = fxpOptimizationOptions;`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `addSpecification(opt,'BlockParameter',bp,'Variable',var)`

`BlockParameter` — Block parameters
`Simulink.Simulation.BlockParameter` object | array of `Simulink.Simulation.BlockParameter` objects

An element or array of `Simulink.Simulation.BlockParameter` objects specifying the data types of block parameters that should not change during the optimization. The value specified must be a valid data type for the block.

`Variable` — Variable values
`Simulink.Simulation.Variable` object | array of `Simulink.Simulation.Variable` objects

An element or array of Simulink.Simulation.Variable objects specifying the data types of variables that should not change during the optimization. You can specify values for Simulink.Parameter or Simulink.NumericType variables.

Examples

Specify Known Data Types for Block Parameters Before Data Type Optimization

This example shows how to specify known data types for block parameters within your system.

Load the system for which you want to optimize the data types.

```
load_system('ex_auto_gain_controller');
```

To specify that the input to the system you are converting will always be an eight-bit integer, create a BlockParameter object that specifies the block parameter, and the data type.

```
bp = Simulink.Simulation.BlockParameter(...
'ex_auto_gain_controller/input_signal','OutDataTypeStr','int8');
```

The fxpOptimizationOptions object, opt, specifies options to use during data type optimization. To specify the data type of the input to the system, use the addSpecification method.

```
opt = fxpOptimizationOptions;
addSpecification(opt,'BlockParameter',bp)
```

You can view all specifications added to a fxpOptimizationOptions object using the showSpecifications method.

```
showSpecifications(opt)
```

```
    Index        Name                     BlockPath                   Value
    _____    ______________    ____________________________________   ______

      1      OutDataTypeStr    ex_auto_gain_controller/input_signal   'int8'
```

Specify Known Data Types for Variables Before Data Type Optimization

This example shows how to specify known data types for variables within your system.

Create a Simulink.Parameter object to set the value a parameter in your model.

```
myParam = Simulink.Parameter(2);
myParamCopy = copy(myParam);
```

Make a copy of the parameter and set the data type to the desired known value.

```
myParamCopy = copy(myParam);
myParamCopy.DataType = 'single';
```

Specify the variable using a Simulink.Simulation.Variable object.

```
var = Simulink.Simulation.Variable('myParam',myParamCopy);
```

The `fxpOptimizationOptions` object, `opt`, specifies options to use during data type optimization. To specify the data type of the variable, use the `addSpecification` method.

```
opt = fxpOptimizationOptions();
addSpecification(opt,'Variable',var);
```

You can view all specifications added to a `fxpOptimizationOptions` object using the `showSpecifications` method.

```
showSpecifications(opt)
```

Version History

Introduced in R2020a

R2021b: Enforce known data types for variables in a system

You can now use the `addSpecification` function to specify known data types for variables within your system in addition to being able to specify known data types for block parameters.

See Also

Classes
`Simulink.Simulation.BlockParameter` | `fxpOptimizationOptions` | `OptimizationResult` | `OptimizationSolution`

Functions
`addTolerance` | `showTolerances` | `explore` | `fxpopt`

Topics
"Optimize Fixed-Point Data Types for a System"
"Optimize Data Types for an FPGA with DSP Slices"

addTolerance

Class: `fxpOptimizationOptions`

Specify numeric tolerance for optimized system

Syntax

```
addTolerance(options,blockPath,portIndex,tolType,tolValue)
addTolerance(options,blockPath,portIndex,tolType,tolValue,
'LoggingInfo',logInfo)
```

Description

`addTolerance(options,blockPath,portIndex,tolType,tolValue)` specifies a numeric tolerance for the output signal specified by `blockPath` and `portIndex`, with the tolerance type specified by `tolType` and value specified by `tolValue`.

`addTolerance(options,blockPath,portIndex,tolType,tolValue, 'LoggingInfo',logInfo)` specifies a tolerance and options for logging information with `Simulink.SimulationData.LoggingInfo`.

Input Arguments

`options` — Associated `fxpOptimizationOptions` object
`fxpOptimizationOptions`

`fxpOptimizationOptions` object to add a tolerance specification.

`blockPath` — Path to block for which to add tolerance
block path name

Path to the block to add a tolerance to, specified as a character vector.

Data Types: `char` | `string`

`portIndex` — Index of output port of block
scalar integer

Index of output port of the block specified by `blockPath` for which you want to specify a tolerance, specified as a scalar integer.

Data Types: `double`

`tolType` — Type of tolerance to specify
`'AbsTol'` | `'RelTol'` | `'TimeTol'`

Type of tolerance to add to the port indicated specified as either absolute tolerance, `'AbsTol'`, relative tolerance, `'RelTol'`, or time tolerance, `'TimeTol'`.

You can specify a tolerance band using a combination of absolute, relative, and time tolerance values. When you specify the tolerance for your signal using multiple types of tolerances, the overall tolerance band is computed by selecting the most lenient tolerance result for each data point.

When you define your tolerance using only the absolute and relative tolerance properties, the tolerance for each point is computed as a simple maximum.

```
tolerance = max(absoluteTolerance,relativeTolerance*abs(baselineData));
```

For detailed information on tolerance computation, see “Tolerance Computation”.

Data Types: `char`

`tolValue` — Difference between the original output and the output of the new design
scalar double

Acceptable level of tolerance for the signal specified by `blockPath` and `portIndex`.

If `tolType` is set to `'AbsTol'`, then `tolValue` represents the absolute value of the maximum acceptable difference between the original output, and the output of the new design.

If `tolType` is set to `'RelTol'`, then `tolValue` represents the maximum relative difference, specified as a percentage, between the original output, and the output of the new design. For example, a value of `1e-2` indicates a maximum difference of one percent between the original output, and the output of the new design.

If `tolType` is set to `'TimeTol'`, then `tolValue` defines a time interval, in seconds, in which the maximum and minimum values define the upper and lower values to compare against.

You can specify a tolerance band using a combination of absolute, relative, and time tolerance values. When you specify the tolerance for your signal using multiple types of tolerances, the overall tolerance band is computed by selecting the most lenient tolerance result for each data point.

When you define your tolerance using only the absolute and relative tolerance properties, the tolerance for each point is computed as a simple maximum.

```
tolerance = max(absoluteTolerance,relativeTolerance*abs(baselineData));
```

For detailed information on tolerance computation, see “Tolerance Computation”.

Data Types: `double`

`'LoggingInfo',logInfo` — Optional signal logging settings
`Simulink.SimulationData.LoggingInfo` object

Optional signal logging settings, specified as a name-value pair where `logInfo` is a `Simulink.SimulationData.LoggingInfo` object. Use this input argument to specify a Decimation value to control the amount of data logged by the Simulation Data Inspector.

Example: `logInfo = Simulink.SimulationData.LoggingInfo(); logInfo.DecimateData = true; logInfo.Decimation = 10; addTolerance(options, 'model/blockPath',2,'AbsTol',1,'LoggingInfo',logInfo);`

Examples

Specify required numeric tolerance for optimized system

Load the system for which you want to optimize the data types.

```
load_system('ex_auto_gain_controller');
```

Create a `fxpOptimizationOptions` object with default property values.

```
options = fxpOptimizationOptions;
```

To specify a required numeric tolerance to use during the optimization process, use the `addTolerance` method of the `fxpOptimizationOptions` object. To specify several tolerance constraints, call the method once per constraint. You can specify either relative, or absolute tolerance constraints.

```
addTolerance(options, 'ex_auto_gain_controller/output_signal', 1, 'AbsTol', 5e-2);
addTolerance(options, 'ex_auto_gain_controller/input_signal', 1, 'RelTol', 1e-2);
```

Use the `showTolerances` method to display all tolerance constraints added to a specified `fxpOptimizationOptions` object.

```
showTolerances(options)
```

```
                   Path                      Port_Index    Tolerance_Type    Tolerance_Value
    _________________________________________    __________    ______________    _______________

    {'ex_auto_gain_controller/output_signal'}        1           {'AbsTol'}           0.05
    {'ex_auto_gain_controller/input_signal' }        1           {'RelTol'}           0.01


ans =

  2x4 table

                   Path                      Port_Index    Tolerance_Type    Tolerance_Value
    _________________________________________    __________    ______________    _______________

    {'ex_auto_gain_controller/output_signal'}        1           {'AbsTol'}           0.05
    {'ex_auto_gain_controller/input_signal' }        1           {'RelTol'}           0.01
```

Version History

Introduced in R2018a

R2021b: Specify multiple types of tolerances

Behavior changed in R2021b

You can now specify multiple types of tolerances using the `addTolerance` function.

```
addTolerance(options,'model/blockPath',1,'AbsTol',5e-2,'RelTol',1e-2);
```

R2021b: Change in syntax for `fxpOptimizationOptions.addTolerance`

Behavior changed in R2021b

In previous releases, you specified options for logging information with a `Simulink.SimulationData.LoggingInfo` object as:

```
addTolerance(options,blockPath,portIndex,tolType,tolValue,loggingInfo)
```

Starting in R2021b, you must now specify logging information as a name-value pair:

```
addTolerance(options,blockPath,portIndex,...
    tolType,tolValue,'LoggingInfo',logInfo)
```

R2021a: Log a reduced set of data points

Using the `addTolerance` method of the `fxpOptimizationOptions` object, you can now control the amount of data logged by the Simulation Data Inspector by specifying a decimation factor.

```
logInfo = Simulink.SimulationData.LoggingInfo();
logInfo.DecimateData = true;
logInfo.Decimation = 10;
addTolerance(options,'model/blockPath',2,'AbsTol',1,logInfo);
```

See Also

Classes
`fxpOptimizationOptions` | `OptimizationResult` | `OptimizationSolution`

Functions
`showTolerances` | `explore` | `fxpopt`

Topics
"Optimize Fixed-Point Data Types for a System"

showSpecifications

Class: fxpOptimizationOptions

Show specifications for a system

Syntax

```
showSpecifications(options)
```

Description

showSpecifications(options) displays all parameters that were specified for a system using the addSpecification method of the fxpOptimizationOptions class. If the options object has no parameters specified, the showSpecifications method does not display anything.

Input Arguments

options — Optimization options
fxpOptimizationOptions object

Optimization options, specified as an fxpOptimizationOptions object with known data types specified for a system.

Examples

Specify Known Data Types for Block Parameters Before Data Type Optimization

This example shows how to specify known data types for block parameters within your system.

Load the system for which you want to optimize the data types.

```
load_system('ex_auto_gain_controller');
```

To specify that the input to the system you are converting will always be an eight-bit integer, create a BlockParameter object that specifies the block parameter, and the data type.

```
bp = Simulink.Simulation.BlockParameter(...
'ex_auto_gain_controller/input_signal','OutDataTypeStr','int8');
```

The fxpOptimizationOptions object, opt, specifies options to use during data type optimization. To specify the data type of the input to the system, use the addSpecification method.

```
opt = fxpOptimizationOptions;
addSpecification(opt,'BlockParameter',bp)
```

You can view all specifications added to a fxpOptimizationOptions object using the showSpecifications method.

```
showSpecifications(opt)
```

```
    Index         Name                          BlockPath                       Value 
    _____    ______________    _____________________________________    ______

      1      OutDataTypeStr    ex_auto_gain_controller/input_signal    'int8'
```

Version History
Introduced in R2020a

See Also

Classes
fxpOptimizationOptions | OptimizationResult | OptimizationSolution

Functions
addTolerance | showTolerances | explore | fxpopt

Topics
"Optimize Fixed-Point Data Types for a System"

showTolerances

Class: fxpOptimizationOptions

Show tolerances specified for a system

Syntax

showTolerances(options)

Description

showTolerances(options) displays the absolute and relative tolerances specified for a system using the addTolerance method of the fxpOptimizationOptions class. If the options object has no tolerances specified, the showTolerances method does not display anything.

Input Arguments

options — Optimization options
fxpOptimizationOptions object

fxpOptimizationOptions object specifying options and tolerances to use during the data type optimization process.

Examples

Specify required numeric tolerance for optimized system

Load the system for which you want to optimize the data types.

```
load_system('ex_auto_gain_controller');
```

Create a fxpOptimizationOptions object with default property values.

```
options = fxpOptimizationOptions;
```

To specify a required numeric tolerance to use during the optimization process, use the addTolerance method of the fxpOptimizationOptions object. To specify several tolerance constraints, call the method once per constraint. You can specify either relative, or absolute tolerance constraints.

```
addTolerance(options, 'ex_auto_gain_controller/output_signal', 1, 'AbsTol', 5e-2);
addTolerance(options, 'ex_auto_gain_controller/input_signal', 1, 'RelTol', 1e-2);
```

Use the showTolerances method to display all tolerance constraints added to a specified fxpOptimizationOptions object.

```
showTolerances(options)
```

```
                   Path                         Port_Index    Tolerance_Type    Tolerance_Value
    ________________________________________    __________    ______________    _______________
```

```
    {'ex_auto_gain_controller/output_signal'}        1          {'AbsTol'}          0.05
    {'ex_auto_gain_controller/input_signal' }        1          {'RelTol'}          0.01


ans =

  2x4 table

                      Path                       Port_Index    Tolerance_Type    Tolerance_Value
    _________________________________________    __________    ______________    _______________

    {'ex_auto_gain_controller/output_signal'}        1          {'AbsTol'}          0.05
    {'ex_auto_gain_controller/input_signal' }        1          {'RelTol'}          0.01
```

Version History

Introduced in R2018a

See Also

Classes
`fxpOptimizationOptions` | `OptimizationResult` | `OptimizationSolution`

Functions
`addTolerance` | `explore` | `fxpopt`

Topics
"Optimize Fixed-Point Data Types for a System"

replace

Replace all Lookup Table blocks with compressed lookup tables

Syntax

```
replace(compressionResult)
replace(compressionResult, index)
```

Description

`replace(compressionResult)` replaces all n-D Lookup Table blocks in a system with the compressed versions described in the `LUTCompressionResult` object `compressionResult`.

`replace(compressionResult, index)` replaces the lookup tables at the indices specified by `index`.

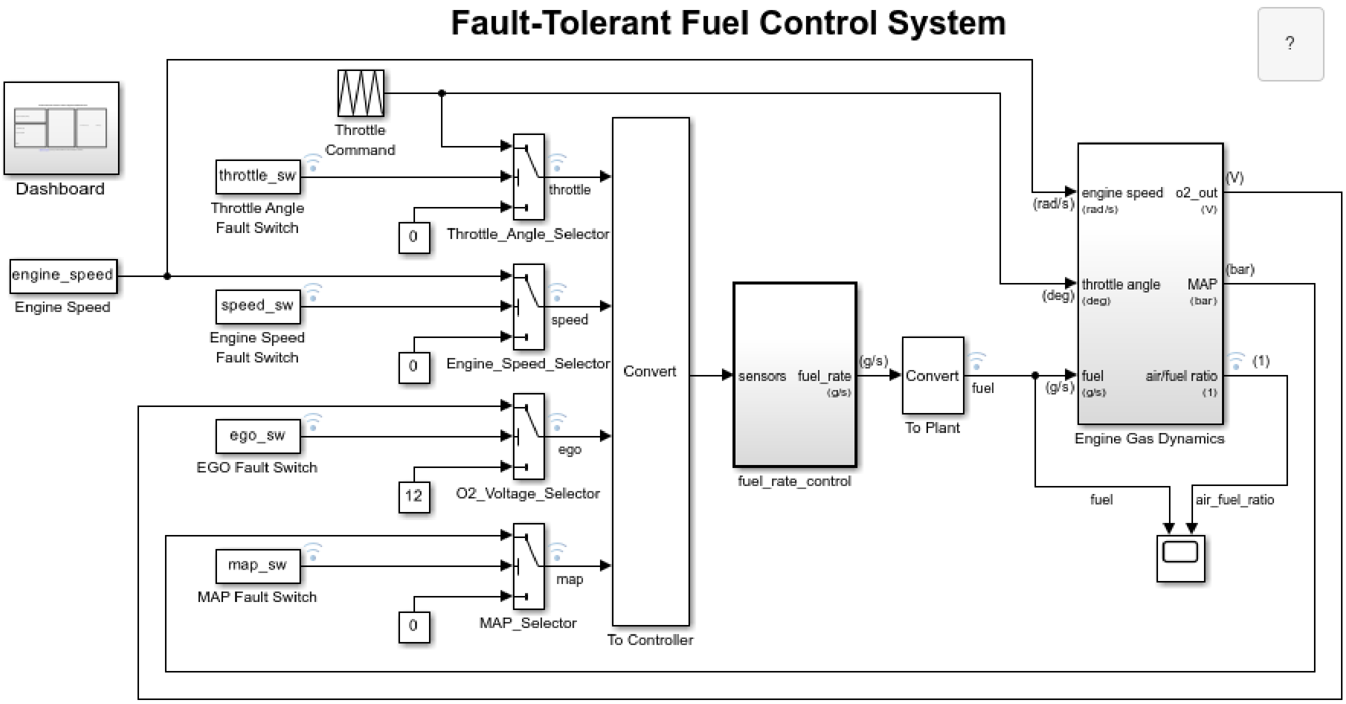
Examples

Compress All Lookup Table Blocks in a System

This example shows how to compress all Lookup Table blocks in a system.

Open the model containing the lookup tables that you want to compress.

```
system = 'sldemo_fuelsys';
open_system(system)
```



Open the Dashboard subsystem to simulate any combination of sensor failures.

Use the `FunctionApproximation.compressLookupTables` function to compress all of the lookup tables in the model. The output specifies all blocks that are modified and the memory savings for each.

```
compressionResult = FunctionApproximation.compressLookupTables(system)
```

```
- Found 5 supported lookup tables
- Percent reduction in memory for compressed solution
    -  2.37% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant
    -  2.37% for sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throt
    -  3.55% for sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estima
    -  6.38% for sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Es
    -  9.38% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki

compressionResult = 

  LUTCompressionResult with properties:

                MemoryUnits: bytes
           MemoryUsageTable: [5x5 table]
               NumLUTsFound: 5
            NumImprovements: 5
            TotalMemoryUsed: 6024
         TotalMemoryUsedNew: 5796
         TotalMemorySavings: 228
  TotalMemorySavingsPercent: 3.7849
                        SUD: 'sldemo_fuelsys'
```

```
WordLengths: [8 16 32]
FindOptions: [1x1 Simulink.internal.FindOptions]
    Display: 1
```

Use the `replace` function to replace each Lookup Table block with a block containing the original and compressed version of the lookup table.

```
replace(compressionResult);
```

You can revert the lookup tables back to their original state using the `revert` function.

```
revert(compressionResult);
```

Input Arguments

`compressionResult` — Results of lookup table compression
`LUTCompressionResult` object

Results of lookup table compression, specified as a `LUTCompressionResult` object.

`index` — Index of Lookup Table blocks to replace
scalar | vector

Index of the Lookup Table blocks to replace in the system, specified as an integer-valued scalar or vector.

The index of each lookup table corresponds to the `ID` column in the `MemoryUsageTable` property of the `LUTCompressionResult` object.

Data Types: `double`

Version History

Introduced in R2020a

See Also

Classes
`LUTCompressionResult`

Functions
`FunctionApproximation.compressLookupTables` | `revert`

revert

Revert compressed Lookup Table blocks to original versions

Syntax

```
revert(compressionResult)
revert(compressionResult, index)
```

Description

`revert(compressionResult)` reverts the Lookup Table blocks compressed by the `FunctionApproximation.compressLookupTables` function back to their original state.

`revert(compressionResult, index)` reverts the lookup tables at the indices specified by `index`.

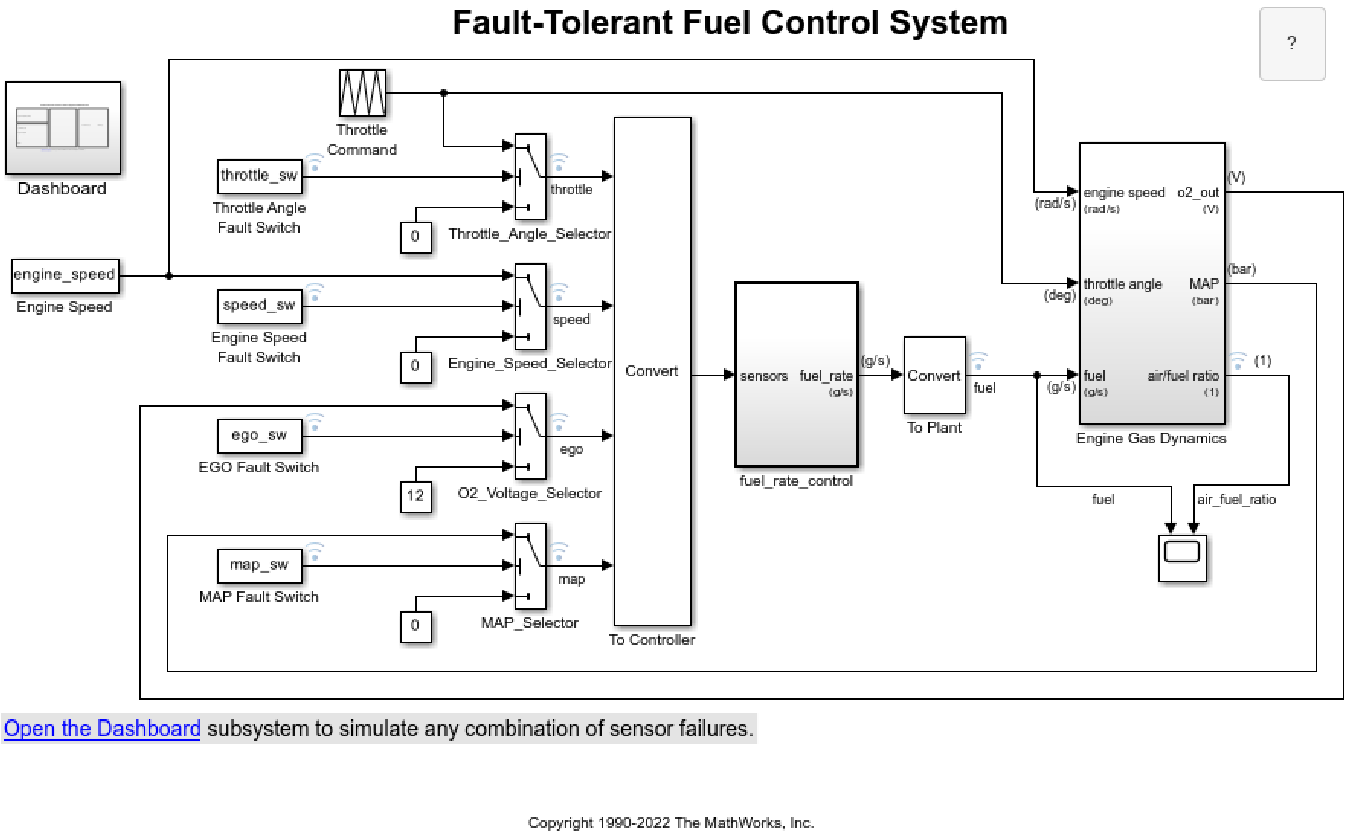
Examples

Compress All Lookup Table Blocks in a System

This example shows how to compress all Lookup Table blocks in a system.

Open the model containing the lookup tables that you want to compress.

```
system = 'sldemo_fuelsys';
open_system(system)
```



Use the `FunctionApproximation.compressLookupTables` function to compress all of the lookup tables in the model. The output specifies all blocks that are modified and the memory savings for each.

```
compressionResult = FunctionApproximation.compressLookupTables(system)
```

```
- Found 5 supported lookup tables
- Percent reduction in memory for compressed solution
    -  2.37% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Pumping Constant
    -  2.37% for sldemo_fuelsys/fuel_rate_control/control_logic/Throttle.throttle_estimate/Throt
    -  3.55% for sldemo_fuelsys/fuel_rate_control/control_logic/Speed.speed_estimate/Speed Estima
    -  6.38% for sldemo_fuelsys/fuel_rate_control/control_logic/Pressure.map_estimate/Pressure Es
    -  9.38% for sldemo_fuelsys/fuel_rate_control/airflow_calc/Ramp Rate Ki

compressionResult = 

  LUTCompressionResult with properties:

                MemoryUnits: bytes
           MemoryUsageTable: [5x5 table]
               NumLUTsFound: 5
            NumImprovements: 5
            TotalMemoryUsed: 6024
         TotalMemoryUsedNew: 5796
         TotalMemorySavings: 228
    TotalMemorySavingsPercent: 3.7849
                        SUD: 'sldemo_fuelsys'
```

```
              WordLengths: [8 16 32]
              FindOptions: [1x1 Simulink.internal.FindOptions]
                  Display: 1
```

Use the `replace` function to replace each Lookup Table block with a block containing the original and compressed version of the lookup table.

```
replace(compressionResult);
```

You can revert the lookup tables back to their original state using the `revert` function.

```
revert(compressionResult);
```

Input Arguments

`compressionResult` — Results of lookup table compression
`LUTCompressionResult` object

Results of lookup table compression, specified as a `LUTCompressionResult` object.

`index` — Index of Lookup Table blocks to revert
scalar | vector

Index of the Lookup Table blocks to revert in the system, specified as an integer-valued scalar or vector.

The index of each lookup table corresponds to the `ID` column in the `MemoryUsageTable` property of the `LUTCompressionResult` object.

Data Types: `double`

Version History

Introduced in R2020a

See Also

Classes
`LUTCompressionResult`

Functions
`FunctionApproximation.compressLookupTables` | `replace`

explore

Class: `OptimizationResult`

Explore fixed-point implementations found during optimization process

Syntax

```
explore(result)
explore(result,Name,Value)
solution = explore(result,Name,Value)
```

Description

`explore(result)` applies the data types of the best solution found during the optimization process for the `OptimizationResult` object specified by `result`. If you have defined tolerances for logged signals in your system, `explore` opens the Simulation Data Inspector with logging data displayed for further exploration of numeric behavior. By default, the best solution and the first simulation scenario will be applied on the model and explored.

`explore(result,Name,Value)` explores `result` with additional options specified by name-value pairs.

`solution = explore(result,Name,Value)` explores `result` with additional options specified by name-value pairs and returns an `OptimizationSolution` object, `solution`.

Input Arguments

`result` — `OptimizationResult` to explore
`OptimizationResult` object

`OptimizationResult` object to explore.

If the optimization finds a feasible solution, the vector of `OptimizationSolution` objects contained in the `result` object is sorted by cost, with the lowest cost (most optimal) solution as the first element of the vector. If the optimization does not find a feasible solution, the vector is sorted by least violation.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `solution = explore(result,'SolutionIndex',1,'ScenarioIndex',5,'KeepOriginalModelParameters',false);`

`SolutionIndex` — n^{th} best solution
1 (default) | positive integer

nth best solution contained in `result` to apply to the model, specified as a positive integer. By default, the best solution is applied.

If optimization finds a feasible result, the best solution is defined as the solution with minimal cost that meets all behavioral constraints. If optimization finds only infeasible solutions, the best solution is defined as the least-violating solution.

Example: `solution = explore(result,'SolutionIndex',2);` returns the second-best solution.

`ScenarioIndex` — nth simulation scenario

1 (default) | positive integer

nth simulation scenario contained in `result`. If no simulation scenarios were used for optimization, this value is set to `1`.

Example: `solution = explore(result,'SolutionIndex',2,'ScenarioIndex',5);` returns the second-best solution using the simulation scenario with index 5.

`KeepOriginalModelParameters` — Whether to maintain original values of model parameters

`false` or `0` (default) | `true` or `1`

Whether to maintain original values of model parameters that are altered during the optimization process, specified as a numeric or logical `1` (`true`) or `0` (`false`).

A value of `true` maintains the original model parameters, but may lead to inconsistencies with the results returned by `fxpopt`. For more information, see "Model Configuration Changes Made During Data Type Optimization".

Example: `solution = explore(result,'KeepOriginalModelParameters',true)` maintains the original values of model parameters.

Output Arguments

`solution` — `OptimizationSolution` containing information related to fixed-point implementation for system

`OptimizationSolution` object

`OptimizationSolution` object containing information related to the optimal fixed-point implementation for the system, including total cost of the implementation and the maximum difference between the baseline and the solution.

Version History

Introduced in R2018a

R2021b: Maintain original values of model parameters that are altered by `fxpopt`

Use the `KeepOriginalModelParameters` option of `explore` to maintain the original values of model configuration parameters during the optimization process.

See Also

Classes
fxpOptimizationOptions | OptimizationResult | OptimizationSolution

Functions
addTolerance | showTolerances | fxpopt

Topics
"Optimize Fixed-Point Data Types for a System"
"Model Configuration Changes Made During Data Type Optimization"

revert

Class: `OptimizationResult`

Revert system data types and settings changed during optimization to original state

Syntax

`revert(result)`

Description

`revert(result)` reverts the changes made during optimization, including system settings and data types, to their original state.

Input Arguments

`result` — `OptimizationResult` to revert
`OptimizationResult` object

`OptimizationResult` object to revert to its state before optimization.

Considerations

If the system you are optimizing contains a MATLAB Function block, the optimization replaces the block with a Variant Subsystem, Variant Model, Variant Assembly Subsystem block in which one variant contains the original MATLAB Function block and the other variant contains the block with the optimized, fixed-point data types. When you revert a system containing a MATLAB Function block, the variant containing the original MATLAB Function block is set as the active variant.

Similarly, if the system you are optimizing contains a Stateflow chart, the optimization process first replaces all data types in the chart with `Simulink.NumericType` objects. When you revert a system containing a Stateflow chart, the data type of the `Simulink.NumericType` objects are restored to their original data type, but the `NumericType` objects still exist in the model.

In both of these cases, when you revert your system, the model behaves numerically identically to how it did before the optimization, however, the model is not actually identical to its state before optimization.

Version History

Introduced in R2020a

See Also

Classes
`fxpOptimizationOptions` | `OptimizationResult` | `OptimizationSolution`

Functions
addTolerance | showTolerances | fxpopt

Topics
"Optimize Fixed-Point Data Types for a System"

openSimulationManager

Class: `OptimizationResult`

Inspect simulations run during optimization in Simulation Manager

Syntax

`openSimulationManager(result)`

Description

`openSimulationManager(result)` opens Simulation Manager with simulations displayed for the `OptimizationResult` object specified by `result`.

Input Arguments

`result` — `OptimizationResult` to inspect
`OptimizationResult`

`OptimizationResult` object containing simulations to inspect in Simulation Manager.

Version History

Introduced in R2020b

See Also

Classes
`fxpOptimizationOptions` | `OptimizationResult` | `OptimizationSolution`

Functions
`addTolerance` | `showTolerances` | `explore` | `revert` | `fxpopt`

Topics
Simulation Manager
“Optimize Fixed-Point Data Types for a System”

showContents

Class: `OptimizationSolution`

Get summary of changes made during data type optimization

Syntax

```
showContents(Solution)
showContents(Solution, index)
```

Description

`showContents(Solution)` returns a summary of the changes made during optimization contained in the `OptimizationSolution` object, `Solution`, including model settings, block parameters, and data types in the model.

`showContents(Solution, index)` returns a summary of the changes made during optimization in the simulation scenario specified by `index`.

Input Arguments

`Solution` — Solution to data type optimization
`OptimizationSolution` object

Solution to data type optimization, specified as an `OptimizationSolution` object.

`index` — Index of simulation scenario
scalar integer

Index of simulation scenario, specified as a scalar integer.

Data Types: `double`

Version History

Introduced in R2020a

See Also

`fxpopt` | `OptimizationSolution`

7

Model Metrics Objects and Object Functions

metric.Engine

Collect metric data on models

Description

A `metric.Engine` object represents the metric engine that you can execute with the `execute` object function to collect metric data on your design. Use the `getMetrics` function to access the metric data and return an array of `metric.Result` objects. Use `generateReport` to access a detailed report of metrics collected. Use design cost metric data to estimate the cost of implementing your design in embedded C code. For additional metrics, see “Model and Code Testing Metrics” (Simulink Check).

Creation

Syntax

```
metric_engine = metric.Engine()
metric_engine = metric.Engine(projectPath)
```

Description

`metric_engine = metric.Engine()` creates a metric engine object that collects metric data on the current project.

`metric_engine = metric.Engine(projectPath)` opens the project `projectPath` and creates a metric engine object that collects metric data on the project.

Input Arguments

`projectPath` — Path of project
character vector | string scalar

Path of project for which to collect metric data, specified as a character vector or string scalar.

Properties

`ProjectPath` — Project for which engine collects metric data
string scalar

This property is read-only.

Project for which engine collects metric data, returned as a string.

Object Functions

deleteMetrics	Delete metric results for model testing artifacts
execute	Collect metric data

generateReport	Generate report file that contains metric results
getArtifactErrors	Return errors that occurred during metric execution
getArtifactIssues	Return issues that occur during artifact analysis
getAvailableMetricIds	Return metric identifiers for available metrics
getMetrics	Access metric data for model testing artifacts
openArtifact	Open testing artifact traced from metric result
updateArtifacts	Update trace information for pending artifact changes in project

Examples

Collect Metric Data for Each Design Unit in Project

Use a `metric.Engine` object to collect design cost metric data on a model reference hierarchy in a project. This example requires Simulink Check™ to run.

To open the project, enter this command.

```
dashboardCCProjectStart
```

The project contains `db_Controller`, which is the top-level model in a model reference hierarchy. This model reference hierarchy represents one design unit.

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect any pending artifact changes.

```
updateArtifacts(metric_engine)
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of all available design cost estimation metrics.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation')
```

```
metric_Ids = 

  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

To collect results, execute the metric engine.

```
execute(metric_engine,metric_Ids);
```

Because the engine was executed without the argument for `ArtifactScope`, the engine collects metrics for the `db_Controller` model reference hierarchy.

Use the `generateReport` function to access detailed metric results in a pdf report. Name the report `'MetricResultsReport.pdf'`.

```
reportLocation = fullfile(pwd,'MetricResultsReport.pdf');
generateReport(metric_engine,...
    'App','DesignCostEstimation',...
```

```
    'Type','pdf',...
    'Location',reportLocation);
```

The report contains a detailed breakdown of the operator count and data segment estimate metric results.

Table of Contents

Version History

Introduced in R2022a

See Also

`getAvailableMetricIds` | `execute` | `generateReport` | `updateArtifacts` | "Design Cost Model Metrics" | "Model Testing Metrics" (Simulink Check)

Topics

"How to Collect Design Cost Metrics"

metric.Result

Metric data for specified metric algorithm

Description

A `metric.Result` object contains the metric data for a specified metric algorithm that traces to the specified unit.

Creation

Syntax

`metric_result = metric.Result`

Description

`metric_result = metric.Result` creates a handle to a metric result object.

Alternatively, if you collect results by executing a `metric.Engine` object, using the `getMetrics` function on the engine object returns the collected `metric.Result` objects in an array.

Properties

`MetricID` — Metric identifier
string

Metric identifier for metric algorithm that calculates results, returned as a string.

Example: `'DataSegmentEstimate'`

`Artifacts` — Testing artifacts
structure | array of structures

Testing artifacts for which metric is calculated, returned as a structure or an array of structures. For each artifact that the metric analyzes, the returned structure contains these fields:

- `UUID` — Unique identifier of artifact
- `Name` — Name of artifact
- `ParentUUID` — Unique identifier of file that contains artifact
- `ParentName` — Name of the file that contains artifact

`Value` — Result value
integer | string | double vector | structure

Result value of the metric for specified algorithm and artifacts, returned as an integer, string, double vector, or structure. For a list of metrics and their result values, see “Design Cost Model Metrics” and “Model Testing Metrics” (Simulink Check).

Scope — Scope of metric results
structure

Scope of metric results, returned as a structure. The scope is the unit for which the metric collected results. The structure contains these fields:

- `UUID` — Unique identifier of unit
- `Name` — Name of unit
- `ParentUUID` — Unique identifier of file that contains unit
- `ParentName` — Name of file that contains unit

`UserData` — User data
string

User data provided by the metric algorithm, returned as a string.

Examples

Collect Metric Data for Each Design Unit in Project

Use a `metric.Engine` object to collect design cost metric data on a model reference hierarchy in a project.

To open the project, enter this command.

```
dashboardCCProjectStart
```

The project contains `cc_CruiseControl`, which is the top-level model in a model reference hierarchy. This model reference hierarchy represents one design unit.

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect any pending artifact changes.

```
updateArtifacts(metric_engine)
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of all available design cost estimation metrics.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation')
```

```
metric_Ids = 

  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

To collect results, execute the metric engine.

```
execute(metric_engine,metric_Ids);
```

Because the engine was executed without the argument for `ArtifactScope`, the engine collects metrics for the `cc_CruiseControl` model reference hierarchy.

Use the `getMetrics` function to access the high-level design cost metric results.

```
results_OperatorCount = getMetrics(metric_engine,'OperatorCount');
results_DataSegmentEstimate = getMetrics(metric_engine,'DataSegmentEstimate');

disp(['Unit:  ', results_OperatorCount.Artifacts.Name])
disp(['Total Cost:  ', num2str(results_OperatorCount.Value)])

disp(['Unit:  ', results_DataSegmentEstimate.Artifacts.Name])
disp(['Data Segment Size (bytes):  ', num2str(results_DataSegmentEstimate.Value)])
```

```
Unit:  cc_CruiseControl
Total Cost:  333

Unit:  cc_CruiseControl
Data Segment Size (bytes):  79
```

The results show that for the `cc_CruiseControl` model, the estimated total cost of the design is 333 and the estimated data segment size is 79 bytes.

Use the `generateReport` function to access detailed metric results in a pdf report. Name the report `'MetricResultsReport.pdf'`.

```
reportLocation = fullfile(pwd,'MetricResultsReport.pdf');
generateReport(metric_engine,...
    'App','DesignCostEstimation',...
    'Type','pdf',...
    'Location',reportLocation);
```

The report contains a detailed breakdown of the operator count and data segment estimate metric results.

Table of Contents

Version History

Introduced in R2022a

See Also

`metric.Engine` | `execute` | `getMetrics` | "Design Cost Model Metrics"

Topics

"How to Collect Design Cost Metrics"

deleteMetrics

Package: metric

Delete metric results for model testing artifacts

Syntax

```
deleteMetrics(metricEngine,metricIDs)
deleteMetrics(metricEngine,metricIDs,'ArtifactScope',scope)
```

Description

deleteMetrics(metricEngine,metricIDs) deletes the metric results specified by metricIDs for the specified metricEngine object. To collect metric results for the metricEngine object, use the execute function. To access the results, use the generateReport function.

deleteMetrics(metricEngine,metricIDs,'ArtifactScope',scope) deletes the metric results for the artifacts in the specified scope. For example, you can specify scope to be a single design unit in your project, such as a Simulink model or an entire model reference hierarchy.

Examples

Delete Metric Data for Specific Metrics

To open the project, enter this command.

```
dashboardCCProjectStart
```

Create a metric.Engine object.

```
metric_engine = metric.Engine();
```

To collect results for the metric OperatorCount, execute the metric engine.

```
execute(metric_engine,{'OperatorCount'});
```

Delete the metric results.

```
deleteMetrics(metric_engine,'OperatorCount')
```

Input Arguments

metricEngine — Metric engine object
metric.Engine object

Metric engine object for which to delete metric results, specified as a metric.Engine object.

metricIDs — Metric identifiers
character vector | cell array of character vectors

Metric identifiers for metrics that you want to delete, specified as a character vector or cell array of character vectors. For a list of design cost metrics, see “Design Cost Model Metrics”. For a list of model testing metrics and their identifiers, see “Model Testing Metrics” (Simulink Check).

Example: `'DataSegmentEstimate'`

Example: `{'DataSegmentEstimate', 'Operator Count'}`

`scope` — Path and identifier of project file
cell array of character vectors

Path and identifier of project file for which to delete metric results, specified as a cell array of character vectors. The first element of the array is the full path to a project file. The second element is the identifier of the object inside the project file.

For a unit model, the first element is the full path to the model file. The second element is the name of the block diagram. When you use this argument, the metric engine deletes the results for the artifacts that trace to specified project file.

Example: `{'C:\work\MyModel.slx', 'MyModel'}`

Tips

- If design changes are not reflected in the design cost metric results, first use the `deleteMetrics` function to delete the `metric.Result`, then use the `execute` function to collect metrics.
- Report generation using the `generateReport` function requires that the metric collection be executed in the current session. To recollect design cost metrics, first use the `deleteMetrics` function to delete the `metric.Result`, then use the `execute` function to collect metrics.

Version History

Introduced in R2022a

See Also

`metric.Engine` | `execute` | `getMetrics` | “Design Cost Model Metrics”

Topics

“How to Collect Design Cost Metrics”

execute

Package: `metric`

Collect metric data

Syntax

```
execute(metricEngine,metricIDs)
execute(metricEngine,metricIDs,'ArtifactScope',scope)
```

Description

`execute(metricEngine,metricIDs)` collects results in the `metricEngine` object specified by `metricEngine` for the metrics specified by `metricIDs`.

`execute(metricEngine,metricIDs,'ArtifactScope',scope)` collects metric results for the artifacts in the specified `scope`. For example, you can specify `scope` to be a single design unit in your project, such as a Simulink model or an entire model reference hierarchy. A unit is a functional entity in your software architecture that you can execute and test independently or as part of larger system tests.

Examples

Collect Metric Data for Each Design Unit in Project

Use a `metric.Engine` object to collect design cost metric data on a model reference hierarchy in a project.

To open the project, enter this command.

```
dashboardCCProjectStart
```

The project contains `cc_CruiseControl`, which is the top-level model in a model reference hierarchy. This model reference hierarchy represents one design unit.

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect any pending artifact changes.

```
updateArtifacts(metric_engine)
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of all available design cost estimation metrics.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation')
```

```
metric_Ids =
```

```
  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

To collect results, execute the metric engine.

```
execute(metric_engine,metric_Ids);
```

Because the engine was executed without the argument for `ArtifactScope`, the engine collects metrics for the `cc_CruiseControl` model reference hierarchy.

Use the `generateReport` function to access detailed metric results in a pdf report. Name the report `'MetricResultsReport.pdf'`.

```
reportLocation = fullfile(pwd,'MetricResultsReport.pdf');
generateReport(metric_engine,...
    'App','DesignCostEstimation',...
    'Type','pdf',...
    'Location',reportLocation);
```

The report contains a detailed breakdown of the operator count and data segment estimate metric results.

Table of Contents

Chapter 1. cc_CruiseControl 1
1.1. Operator Count 2
1.1.1. High Level Statistics 2
1.1.2. Cost Breakdown Details 3
1.2. Data Segment Table 34

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for which to collect metric results, specified as a `metric.Engine` object.

`metricIDs` — Metric identifiers
character vector | cell array of character vectors

Metric identifiers for metrics to collect, specified as a character vector or cell array of character vectors. Collecting results for design cost metrics requires a Fixed-Point Designer license. For a list of design cost metrics and their identifiers, see “Design Cost Model Metrics”. For additional metrics, see “Model and Code Testing Metrics” (Simulink Check).

Example: `'DataSegmentEstimate'`

Example: `{'DataSegmentEstimate', 'OperatorCount'}`

`scope` — Path and identifier of project file
cell array of character vectors

Path and identifier of project file for which to execute metric results, specified as a cell array of character vectors. The first entry is the full path to a project file. The second entry is the identifier of the object inside the project file.

For a unit model, the first entry is the full path to the model file. The second entry is the name of the block diagram. When you use this argument, the metric engine executes the metrics for the artifacts that trace to specified project file.

Example: `{'C:\work\MyModel.slx', 'MyModel'}`

Version History

Introduced in R2022a

See Also

`metric.Engine` | `getAvailableMetricIds` | `generateReport` | `updateArtifacts` | “Design Cost Model Metrics” | “Model Testing Metrics” (Simulink Check)

Topics

“How to Collect Design Cost Metrics”

generateReport

Package: metric

Generate report file that contains metric results

Syntax

```
reportFile = generateReport(metricEngine,'App','DesignCostEstimation')
reportFile = generateReport( ___ ,Name,Value)
```

Description

`reportFile = generateReport(metricEngine,'App','DesignCostEstimation')` creates a PDF report of the metric results from `metricEngine` in the root folder of the project. The generated report shows detailed design cost metric results. Before you generate the report, collect metric results for the engine by using the `execute` function. For a syntax to generate a report for requirements-based model metrics, see `generateReport` (Simulink Check).

`reportFile = generateReport( ___ ,Name,Value)` specifies options using one or more name-value arguments. For example, `'Type','html-file'` generates an HTML file.

Examples

Collect Metric Data for Each Design Unit in Project

Use a `metric.Engine` object to collect design cost metric data on a model reference hierarchy in a project.

To open the project, enter this command.

```
dashboardCCProjectStart
```

The project contains `cc_CruiseControl`, which is the top-level model in a model reference hierarchy. This model reference hierarchy represents one design unit.

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect any pending artifact changes.

```
updateArtifacts(metric_engine)
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of all available design cost estimation metrics.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation')
```

```
metric_Ids =
```

```
  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

To collect results, execute the metric engine.

```
execute(metric_engine,metric_Ids);
```

Because the engine was executed without the argument for `ArtifactScope`, the engine collects metrics for the `cc_CruiseControl` model reference hierarchy.

Use the `generateReport` function to access detailed metric results in a pdf report. Name the report `'MetricResultsReport.pdf'`.

```
reportLocation = fullfile(pwd,'MetricResultsReport.pdf');
generateReport(metric_engine,...
    'App','DesignCostEstimation',...
    'Type','pdf',...
    'Location',reportLocation);
```

The report contains a detailed breakdown of the operator count and data segment estimate metric results.

Table of Contents

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for which metric results are collected, specified as a `metric.Engine` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Type','html-file'`

`LaunchReport` — Whether to open generated report automatically
`true` (default) | `false`

Whether to open generated report automatically, specified as `true` or `false`.

Data Types: `logical`

`Location` — Full file name
character vector | string scalar

Full file name for generated report, specified as a character vector or string scalar. Use the location to specify the name of the report. By default, the report is named `untitled`.

Example: `'C:\MyProject\Reports\RBTResults.html'`

Type — File type
`'pdf'` (default) | `'html-file'`

File type for generated report, specified as `'pdf'` or `'html-file'`.

Example: `'html-file'`

Output Arguments

`reportFile` — Full file name of generated report
character vector

Full file name of generated report, returned as a character vector.

Version History

Introduced in R2022a

See Also

`metric.Engine` | `getAvailableMetricIds` | `execute` | `updateArtifacts` | “Design Cost Model Metrics” | “Model Testing Metrics” (Simulink Check)

Topics
“How to Collect Design Cost Metrics”

getArtifactErrors

Package: metric

Return errors that occurred during metric execution

Syntax

```
errors = getArtifactErrors(metricEngine)
```

Description

`errors = getArtifactErrors(metricEngine)` returns the errors that occur when the `metricEngine` analyzes the Simulink models. The `metricEngine` object does not collect results for artifacts that return errors during analysis.

Examples

Check for Artifact Errors After Collecting Metric Results

Collect design cost metrics for artifacts in a project. Then, check if artifacts return errors and were not analyzed.

To open the project, enter this command.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the design cost metrics by using the `execute` function on the `metric.Engine` object.

```
execute(metric_engine,{'DataSegmentEstimate','OperatorCount'});
```

Access the errors that occurred during analysis.

```
getArtifactErrors(metric_engine)
```

```
ans = 

  0×0 empty struct array with fields:

    Address
    UUID
    ErrorId
    ErrorMessage
```

For this example, the artifacts did not return errors.

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object to check for errors, specified as a `metric.Engine` object.

Output Arguments

`errors` — Artifact errors
`struct` array

Artifact errors that occur when `metric.Engine` object is executed, returned as an array of structures that correspond to the errors. The structure for an error contains these fields:

- `Address` — Address of artifact that returns the error
- `UUID` — Unique identifier of artifact
- `ErrorID` — Identifier of error
- `ErrorMessage` — Description of error

Version History

Introduced in R2022a

See Also

`metric.Engine` | `execute` | `getMetrics` | `updateArtifacts` | `getArtifactIssues` | “Design Cost Model Metrics”

Topics
“How to Collect Design Cost Metrics”

getArtifactIssues

Package: `metric`

Return issues that occur during artifact analysis

Syntax

`issues = getArtifactIssues(metricEngine)`

Description

`issues = getArtifactIssues(metricEngine)` returns the artifact issues that occur when the metric engine analyzes the artifacts in the project. For the metric engine to perform artifact analysis and check for artifact issues, collect metric results by using the `execute` function. If the `execute` function returns a warning about artifact issues in the project, use `getArtifactIssues` to get a list of the artifact issues that the metric engine found during artifact analysis.

If there are artifact issues in your project, try to fix the issues to prevent the artifacts from returning incorrect metric results.

Examples

Identify Artifact Issues After Collecting Metric Results

Collect metric results for the testing artifacts and get a list of artifact issues in the project.

Open the project that contains the models and testing artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Collect results for the design cost metrics by using the `execute` function on the `metric.Engine` object.

```
execute(metric_engine,{'DataSegmentEstimate','OperatorCount'});
```

If there are artifact issues in the project, the `execute` function returns a warning. For this example, the `execute` function does not return a warning.

You can use the function `getArtifactIssues` to get a list of the artifact issues that the metric engine found during artifact analysis.

```
issues = getArtifactIssues(metric_engine)
```

```
issues = 

  0×0 empty struct array with fields:
```

```
    IssueId
    IssueMessage
    Severity
    Address
    UUID
```

For this example, the artifacts did not return any issues. If there are artifact issues in your project, try to fix the issues to prevent the artifacts from returning incorrect metric results.

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for the project for which you want the list of artifact issues, specified as a `metric.Engine` object.

Output Arguments

`issues` — Artifact issues in project
`struct` | `struct` array

Artifact issues in a project, returned as a `struct` or `struct` array with fields:

- `IssueId` — Identifier for the artifact issue type
- `IssueMessage` — Message that describes the artifact issue
- `Severity` — Severity of the artifact issue
- `Address` — Address of the affected artifact
- `UUID` — Universal unique identifier for the affected artifact

There is one element in the array for each artifact issue.

Version History

Introduced in R2023a

See Also

`metric.Engine` | `execute` | `getMetrics` | `updateArtifacts` | `getArtifactErrors` | "Design Cost Model Metrics"

Topics

"How to Collect Design Cost Metrics"

getAvailableMetricIds

Return metric identifiers for available metrics

Syntax

```
availableMetricIds = getAvailableMetricIds(metricEngine)
availableMetricIds = getAvailableMetricIds(
metricEngine,'App','DesignCostEstimation')
availableMetricIds = getAvailableMetricIds( ___ ,'Installed',
installationStatus)
```

Description

`availableMetricIds = getAvailableMetricIds(metricEngine)` returns the metric identifiers for the metrics available for the specified `metricEngine` object. By default, the list includes only the metrics available with the current installation.

`availableMetricIds = getAvailableMetricIds(metricEngine,'App','DesignCostEstimation')` returns the metric identifiers for design cost estimation metrics. For an additional syntax to display metric identifiers for requirements-based model metrics, see `getAvailableMetricIds`.

`availableMetricIds = getAvailableMetricIds( ___ ,'Installed',installationStatus)` returns the metric identifiers filtered by the installation status specified by `installationStatus`. For example, specifying `installationStatus` as `false` returns the metric identifier for each available metric, even if the associated MathWorks products are not currently installed on your machine.

Examples

View Available Metrics

Create a `metric.Engine` object and view all metrics available with the current installation.

```
metric_engine = metric.Engine();
ids = getAvailableMetricIds(metric_engine);
```

View Available Design Cost Metrics

Create a `metric.Engine` object and view all design cost metrics available.

```
metric_engine = metric.Engine();
ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation',...
    'Installed',false)
```

```
ids =
```

```
  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for which to collect metric results, specified as a `metric.Engine` object.

`installationStatus` — Filter for metric installation status
`1` (`true`) (default) | `0` (`false`)

Filter for metric installation status, specified as one of these values:

- `1` (`true`) — Returns only metric identifiers associated with the MathWorks products currently installed on your machine.
- `0` (`false`) — Returns metric identifiers for each available metric, even if the associated MathWorks products are not currently installed on your machine.

Example: `false`

Data Types: `logical`

Output Arguments

`availableMetricIds` — Metric identifiers
string | string array

Metric identifiers for available metrics, returned as a string or string array. For a list of design cost metrics and their identifiers, see “Design Cost Model Metrics”. For a list of requirements-based model testing metrics and their identifiers, see “Model Testing Metrics” (Simulink Check).

Example: `"DataSegmentEstimate"`

Example: `["DataSegmentEstimate","OperatorCount"]`

Version History

Introduced in R2022a

See Also

`metric.Engine` | `execute` | `generateReport` | `updateArtifacts` | “Design Cost Model Metrics” | “Model Testing Metrics” (Simulink Check)

Topics
“How to Collect Design Cost Metrics”

getMetrics

Package: `metric`

Access metric data for model testing artifacts

Syntax

```
results = getMetrics(metricEngine,metricIDs)
results = getMetrics(metricEngine,metricIDs,'ArtifactScope',scope)
```

Description

`results = getMetrics(metricEngine,metricIDs)` returns metric results for the specified `metric.Engine` object for the metrics specified by `metricIDs`. To collect metric results for the `metricEngine` object, use the `execute` function. Then, to access the results, use the `getMetrics` function.

`results = getMetrics(metricEngine,metricIDs,'ArtifactScope',scope)` returns metric results for the artifacts in the specified `scope`. For example, you can specify `scope` to be a single design unit in your project, such as a Simulink model or an entire model reference hierarchy. A unit is a functional entity in your software architecture that you can execute and test independently or as part of larger system tests.

Examples

Collect Metric Data for Each Design Unit in Project

Use a `metric.Engine` object to collect design cost metric data on a model reference hierarchy in a project.

To open the project, use this command.

```
dashboardCCProjectStart
```

The project contains `cc_CruiseControl`, which is the top-level model in a model reference hierarchy. This model reference hierarchy represents one design unit.

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect any pending artifact changes.

```
updateArtifacts(metric_engine)
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of all available design cost estimation metrics.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation')
```

```
metric_Ids = 

  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

To collect results, execute the metric engine.

```
execute(metric_engine,metric_Ids);
```

Because the engine was executed without the argument for `ArtifactScope`, the engine collects metrics for the `cc_CruiseControl` model reference hierarchy.

Use the `getMetrics` function to access the high-level design cost metric results.

```
results_OpCount = getMetrics(metric_engine,'OperatorCount');
results_DataSegmentEstimate = getMetrics(metric_engine,'DataSegmentEstimate');

disp(['Unit:  ', results_OpCount.Artifacts.Name])
disp(['Total Cost:  ', num2str(results_OpCount.Value)])

disp(['Unit:  ', results_DataSegmentEstimate.Artifacts.Name])
disp(['Data Segment Size (bytes):  ', num2str(results_DataSegmentEstimate.Value)])
```

```
Unit:  cc_CruiseControl
Total Cost:  333

Unit:  cc_CruiseControl
Data Segment Size (bytes):  79
```

The results show that for the `cc_CruiseControl` model, the estimated total cost of the design is 333 and the estimated data segment size is 79 bytes.

Use the `generateReport` function to access detailed metric results in a pdf report. Name the report `'MetricResultsReport.pdf'`.

```
reportLocation = fullfile(pwd,'MetricResultsReport.pdf');
generateReport(metric_engine,...
    'App','DesignCostEstimation',...
    'Type','pdf',...
    'Location',reportLocation);
```

The report contains a detailed breakdown of the operator count and data segment estimate metric results.

Table of Contents

Input Arguments

metricEngine — Metric engine object
metric.Engine object

Metric engine object for which to access metric results, specified as a `metric.Engine` object.

metricIDs — Metric identifiers
character vector | cell array of character vectors

Metric identifiers for metrics to access, specified as a character vector or cell array of character vectors. For a list of design cost metrics and their identifiers, see “Design Cost Model Metrics”. For a list of requirements-based model testing metrics and their identifiers, see “Model Testing Metrics” (Simulink Check).

Example: `'DataSegmentEstimate'`

Example: `{'DataSegmentEstimate', 'OperatorCount'}`

scope — Path and identifier of project file
cell array of character vectors

Path and identifier of project file for which to get metric results, specified as a cell array of character vectors. The first entry is the full path to a project file. The second entry is the identifier of the object inside the project file.

For a unit model, the first entry is the full path to the model file. The second entry is the name of the block diagram.

Example: `{'C:\work\MyModel.slx', 'MyModel'}`

Output Arguments

results — Metric results
array of `metric.Result` objects

Metric results, returned as an array of `metric.Result` objects.

Version History

Introduced in R2022a

See Also

`metric.Engine` | `getAvailableMetricIds` | `execute` | `generateReport` | `updateArtifacts` | "Design Cost Model Metrics" | "Model Testing Metrics" (Simulink Check)

Topics

"How to Collect Design Cost Metrics"

openArtifact

Package: metric

Open testing artifact traced from metric result

Syntax

```
openArtifact(metricEngine,artifactID)
```

Description

openArtifact(metricEngine,artifactID) opens the artifact that has the specified identifier artifactID in the specified metricEngine object. The editor that opens depends on the type of artifact.

- Simulink models open in the Simulink Editor.
- Requirements open in the Requirements Editor.
- Test cases and test results open in the Test Manager.

Examples

Open Model Artifact from Metric Result

Use a metric.Engine object to collect design cost metric data on a model reference hierarchy in a project. Then, open one of the top-level model in the Simulink editor.

To open the project, enter this command.

```
dashboardCCProjectStart
```

Create a metric.Engine object.

```
metric_engine = metric.Engine();
```

Update the trace information for metric_engine to reflect any pending artifact changes and ensure that all test results are tracked.

```
updateArtifacts(metric_engine)
```

To collect results for the metric OperatorCount, execute the metric engine.

```
execute(metric_engine,{'OperatorCount'});
```

Use the getMetrics function to access the results.

```
results = getMetrics(metric_engine,'OperatorCount');
disp(['Unit:  ', results.Artifacts.Name])
disp(['Total Cost:  ', num2str(results.Value)])
```

```
Unit:  cc_CruiseControl
Total Cost:  333
```

Open the model artifact in the Simulink Editor by using the artifact identifier.

```
openArtifact(metric_engine,results(1).Artifacts(1).UUID)
```

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for which metric results are collected, specified as a `metric.Engine` object.

`artifactID` — Artifact identifier
character vector | string scalar

Artifact identifier, specified as a character vector or string scalar. In a `metric.Result` object, the `Artifacts` field contains a structure for each artifact to which the result traces. To get the identifier for an artifact, use the `UUID` field of the structure for the artifact.

Version History

Introduced in R2022a

See Also

`metric.Engine` | `execute` | `getMetrics` | "Design Cost Model Metrics"

Topics
"How to Collect Design Cost Metrics"

updateArtifacts

Update trace information for pending artifact changes in project

Syntax

```
updateArtifacts(metricEngine)
```

Description

`updateArtifacts(metricEngine)` updates the trace information for any pending artifact changes in the metric data specified by `metricEngine` to ensure that artifacts are captured by the metrics. If an artifact has been created, deleted, or modified since the last time you used `updateArtifacts`, running `updateArtifacts` performs traceability analysis and updates the trace information.

Examples

Collect Metric Data for Each Design Unit in Project

Use a `metric.Engine` object to collect design cost metric data on a model reference hierarchy in a project.

To open the project, enter this command.

```
dashboardCCProjectStart
```

The project contains `cc_CruiseControl`, which is the top-level model in a model reference hierarchy. This model reference hierarchy represents one design unit.

Create a `metric.Engine` object.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect any pending artifact changes.

```
updateArtifacts(metric_engine)
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of all available design cost estimation metrics.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
    'App','DesignCostEstimation')
```

```
metric_Ids = 

  1×2 string array

    "DataSegmentEstimate"    "OperatorCount"
```

To collect results, execute the metric engine.

```
execute(metric_engine,metric_Ids);
```

Because the engine was executed without the argument for `ArtifactScope`, the engine collects metrics for the `cc_CruiseControl` model reference hierarchy.

Use the `generateReport` function to access detailed metric results in a pdf report. Name the report `'MetricResultsReport.pdf'`.

```
reportLocation = fullfile(pwd,'MetricResultsReport.pdf');
generateReport(metric_engine,...
    'App','DesignCostEstimation',...
    'Type','pdf',...
    'Location',reportLocation);
```

The report contains a detailed breakdown of the operator count and data segment estimate metric results.

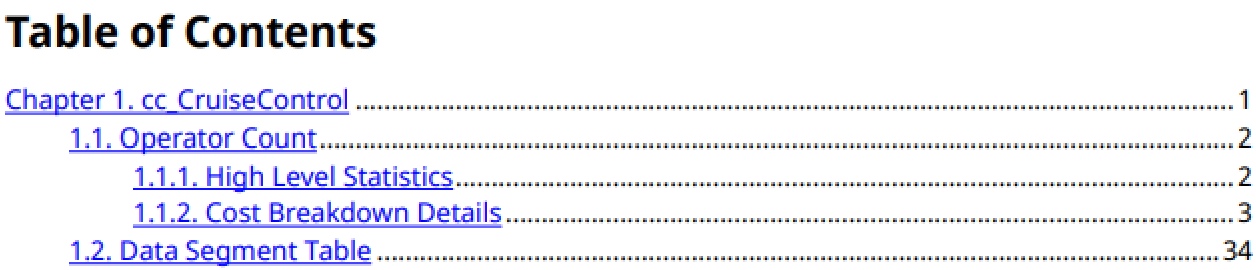


Table of Contents

Chapter 1. cc_CruiseControl 1
1.1. Operator Count 2
1.1.1. High Level Statistics 2
1.1.2. Cost Breakdown Details 3
1.2. Data Segment Table 34

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for which to collect metric results, specified as a `metric.Engine` object.

Version History

Introduced in R2022a

See Also

`metric.Engine` | `getAvailableMetricIds` | `execute` | `generateReport` | "Design Cost Model Metrics" | "Model Testing Metrics" (Simulink Check)

Topics

"How to Collect Design Cost Metrics"

Selected Bibliography

[1] Burrus, C.S., J.H. McClellan, A.V. Oppenheim, T.W. Parks, R.W. Schafer, and H.W. Schuessler, *Computer-Based Exercises for Signal Processing Using MATLAB*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[2] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems, Second Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[3] *Handbook For Digital Signal Processing*, edited by S.K. Mitra and J.F. Kaiser, John Wiley & Sons, Inc., New York, 1993.

[4] Hanselmann, H., "Implementation of Digital Controllers — A Survey," *Automatica*, Vol. 23, No. 1, pp. 7-32, 1987.

[5] Jackson, L.B., *Digital Filters and Signal Processing, Second Edition*, Kluwer Academic Publishers, Seventh Printing, Norwell, Massachusetts, 1993.

[6] Middleton, R. and G. Goodwin, *Digital Control and Estimation — A Unified Approach*, Prentice Hall, Englewood Cliffs, New Jersey. 1990.

[7] Moler, C., "Floating points: IEEE Standard unifies arithmetic model," Cleve's Corner, The MathWorks, Inc., 1996. You can find this article at `https://www.mathworks.com/company/newsletters/news_notes/clevescorner/index.html`.

[8] Ogata, K., *Discrete-Time Control Systems*, *Second Edition*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[9] Roberts, R.A. and C.T. Mullis, *Digital Signal Processing*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.