

Navigation Toolbox™ Release Notes



MATLAB® & SIMULINK®



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

Navigation Toolbox™ Release Notes

© COPYRIGHT 2019–2021 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

R2021b

Motion Planning Using Control-Based RRT	1-2
Visualize Rigid Body Pose	1-2
Simultaneous Localization and Mapping (SLAM) Using Extended Kalman Filter (EKF)	1-2
GPS Sensor Block	1-3
Bicycle Vehicle Model	1-3
Hybrid A* Path Planner Update	1-3
Perturb IMU sensor readings	1-3
NMEA Parser Enhancements	1-3

R2021a

Multigraph and Landmark Support For poseGraph and poseGraph3D ...	2-2
poseGraph and poseGraph3D Object Functions Renamed	2-2
Multilayer Grid Maps: Sync multiple map layers and store additional dynamic information	2-2
Global Navigation Satellite System Simulation Calculations: Access GNSS constellations, satellite look angles, pseudoranges, and receiver positions	2-2
Sky Plot Visualization: Plot and label satellite positions in a polar plot	2-2
Comprehensive support for tuning inertial sensor filters	2-3
Bidirectional RRT Path Planner: Plan path using the bidirectional rapidly exploring random tree algorithm	2-4
Generate Random Maze Maps for Planner Benchmarking	2-4

Transform Between Geodetic Coordinates and Local Cartesian Coordinates	2-5
RRT* Path Planner Improvements: Returns PathCosts field in the solution information structure	2-6
Lidar SLAM Object Update: Specify scan registration method to use while adding scan	2-6
optimizePoseGraph Function Update	2-6
NMEA Parser System Object Update: Support for GSV sentences	2-6
updateOccupancy Object Function Update	2-6
Use geodetic coordinates as inputs to gpsSensor	2-6
insSensor provides more properties to specify its characteristics	2-6
Variable-sized input support for timescope object	2-7

R2020b

Grid-Based A* Path Planning: Plan path from start to goal location using the A* algorithm	3-2
Trajectory Optimal Frenet Utilities: More control in generating optimal trajectory in Frenet space	3-2
Dynamic Capsule-Based Obstacle: Model and simulate ego bodies and obstacles using collision primitive objects in environment	3-3
SLAM Map Builder Update on Data Import: Import lidar scans and odometry data from MATLAB workspace	3-3
Pose Graph Optimization Updates: Additional optimization functionality for pose graphs	3-3
Wheel Encoder Sensor Models: Simulate wheel encoder sensor readings	3-3
Wheel Encoder Odometry: Compute vehicle odometry using wheel encoder sensor readings	3-3
INS Sensor Model: Simulate inertial navigation and GPS readings	3-4
GNSS Sensor Model: Simulate GNSS receiver readings	3-4
Code Generation for 3-D Occupancy Map: Generate C/C++ code using occupancyMap3D object	3-4

State Space and State Validator for 3-D Occupancy Map	3-4
Generate Random Grid Maps for Planner Benchmarking	3-4
Adjust Inertial Sensor Fusion Performance Using Filter Tuner	3-5
GPS Device Object: Connect to GPS receiver from host computer	3-5
NMEA Parser Object: Parse data from standard NMEA sentences sent from GNSS receivers	3-6
Time Scope object: Bilevel measurements, triggers, and compiler support	3-6
New examples	3-6

R2020a

New Time Scope object: Visualize signals in the time domain	4-2
Scope Tab	4-2
Measurements Tab	4-2
Scale Axes	4-3
Scan Matching Using Line Features: Estimate pose and covariance based on line features in lidar scans	4-3
Trajectory Optimization Improvements: Specify longitudinal segments, deviation offsets, and additional waypoint parameters	4-3
Path Metrics Improvements: Specify validatorVehicleCostmap as a state validator	4-4
Ray Intersections for 3-D Maps: Calculate ray intersections, import, and export with a 3-D occupancy map	4-4
Code Generation for Monte Carlo Localization: Generate C/C++ code using the monteCarloLocalization object	4-4
Code Generation for Sampling-Based Planners: Generate C/C++ code using the plannerRRT, plannerRRTStar, and plannerHybridAStar objects	4-4
Code Generation for Trajectory Optimization: Generate C/C++ code using the trajectoryOptimalFrenet object	4-4
Access residuals and residual covariance of insfilters and ahrs10filter	4-4
Model inertial measurement unit using IMU Simulink block	4-4

Estimate device orientation using AHRS Simulink block	4-4
Calculate angular velocity from quaternions	4-5
Transform position and velocity between two frames to motion quantities in a third frame	4-5

R2019b

Simultaneous Localization and Mapping (SLAM): Create 2-D and 3-D occupancy maps using SLAM algorithm and lidar scan data	5-2
SLAM Map Builder App: Interactively modify loop closures and adjust overall map using SLAM algorithm	5-2
Pose Estimation: Accurately estimate vehicle poses using IMU and GPS sensors and Monte Carlo Localization	5-2
Customizable Sampling-Based Path Planners: Plan a path from start to goal locations using RRT and RRT* algorithms	5-2
Path-Planning Metrics: Use metrics to check and compare the output of path planners	5-3
Sensor Models: Use simulated models for IMU, GPS, and range sensors	5-3
Trajectory and Waypoint Following Algorithms: Use built-in algorithms to generate trajectories and control commands for robots	5-3

R2021b

Version: 2.1

New Features

Bug Fixes

Motion Planning Using Control-Based RRT

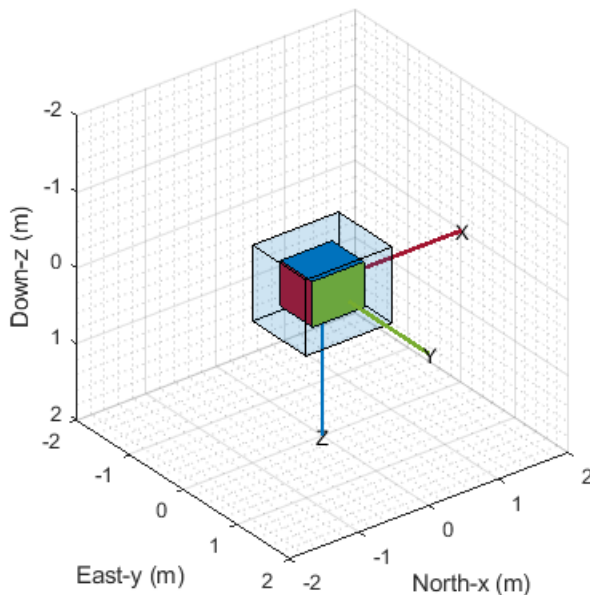
The `plannerControlRRT` object plans a series of control commands using the rapidly exploring random tree (RRT) algorithm. The `plan` function incrementally searches for feasible states and connects them to a tree until a valid path to the goal state is found. The state propagator generates control commands and durations that connect each state in the path. The `navPathControl` object stores the states, controls, and durations for the planned path.

The `nav.StatePropagator` class enables you to implement custom control behavior for state spaces defined by an instance of the `nav.StateSpace` class. You can specify control limits for your system, propagate the system states for given controls, and sample controls for desired target states, as well as estimate the distance cost for control paths.

The `mobileRobotPropagator` object provides a specific implementation of the state propagator for mobile robots that uses a bicycle kinematic model, or accepts other wheel-based motion models from Robotics System Toolbox™.

Visualize Rigid Body Pose

Visualize a rigid body pose (position and orientation) using the `poseplot` function. To customize the appearance of the pose plot, see PosePatch Properties.



Simultaneous Localization and Mapping (SLAM) Using Extended Kalman Filter (EKF)

Use the `ekfSLAM` object to implement EKF-based landmark SLAM. The object takes in observed landmarks from the environment and compares them with the known landmarks in the 2-D map to find the associations and new landmarks. The object uses the associations to correct the state and state covariance. The new landmarks are augmented in the state vector.

GPS Sensor Block

Use the GPS block to simulate a GPS sensor with noise-corrupted measurements based on the input position and velocity. The block uses the WGS84 earth model to convert local coordinates to latitude-longitude-altitude (LLA) coordinates. You can vary the horizontal position, vertical position, and velocity accuracies to change noise levels in the output signals. Also, you can specify a decay factor, which models global position noise.

Bicycle Vehicle Model

Use the `bicycleKinematics` vehicle model to simulate simplified car-like vehicle dynamics.

Hybrid A* Path Planner Update

The `plan` function of the `plannerHybridAStar` object now returns the direction of motion directions for each pose along the path as well as a solution information structure `solutionInfo` containing the fields `IsPathFound`, `NumNodes`, and `NumIterations`.

Perturb IMU sensor readings

You can now use the `perturbations` and `perturb` functions to perturb IMU sensor readings generated by the `imuSensor` System object™. By perturbing the IMU, you can perform noise analysis to test your fusion filter with different IMU parameters and noise ranges.

NMEA Parser Enhancements

You can now configure the `nmeaParser` System object to extract National Marine Electronics Association (NMEA) data by using a name-value pair, `CustomSentence`. This enhancement enables you to configure the `nmeaParser` System object to extract data from the standard sentences defined as part of the NMEA 0183® Standard, Version 4.10, in addition to the built-in support for certain sentences. You can also use this name-value pair to configure the `nmeaParser` System object to extract data from manufacturer-specific sentences that are recognized by the NMEA.

Use the `extractNMEASentence` function to validate the checksum of the NMEA sentence that you want to parse, and include it in a function file to convert the NMEA data into string array. You can then pass the specific `MessageID` and the name of its corresponding function, as a name-value argument, to the `CustomSentence` name-value pair to obtain the parsed data.

R2021a

Version: 2.0

New Features

Bug Fixes

Compatibility Considerations

Multigraph and Landmark Support For poseGraph and poseGraph3D

The poseGraph and poseGraph3D objects have been updated to support adding multiple edges between nodes and adding landmark points to a pose graph. The addRelativePose function now appends a new edge between specific nodes instead of overwriting the existing edge, which allows one to incorporate multiple sensor readings for node pose estimation. To add point landmarks, use the addPointLandmark function.

poseGraph and poseGraph3D Object Functions Renamed

These poseGraph and poseGraph3D object functions have been renamed to better reflect their definitions:

Function Name in R2020b and Earlier	Function Name in R2021a
edges	edgeNodePairs
nodes	nodeEstimates

Multilayer Grid Maps: Sync multiple map layers and store additional dynamic information

The multiLayerMap object groups and stores multiple map layers as binaryOccupancyMap, occupancyMap, or mapLayer objects. Once added, the map layers can be modified together using the multiLayerMap object functions. Individual actions applied to each map are synced with the multilayer map.

Assign and retrieve data from one or more cells in the map using the setMapData and getMapData object functions. Additionally, mapLayer objects can store numeric N -dimensional arrays, enabling you to store more than just occupancy values in each cell. This feature enables dynamic tracking of obstacles and other more advanced applications for mapping.

Sync data across maps using the syncWith function. Move the map in the world frame using the move function. Convert between grid, local, and world coordinates with other object functions.

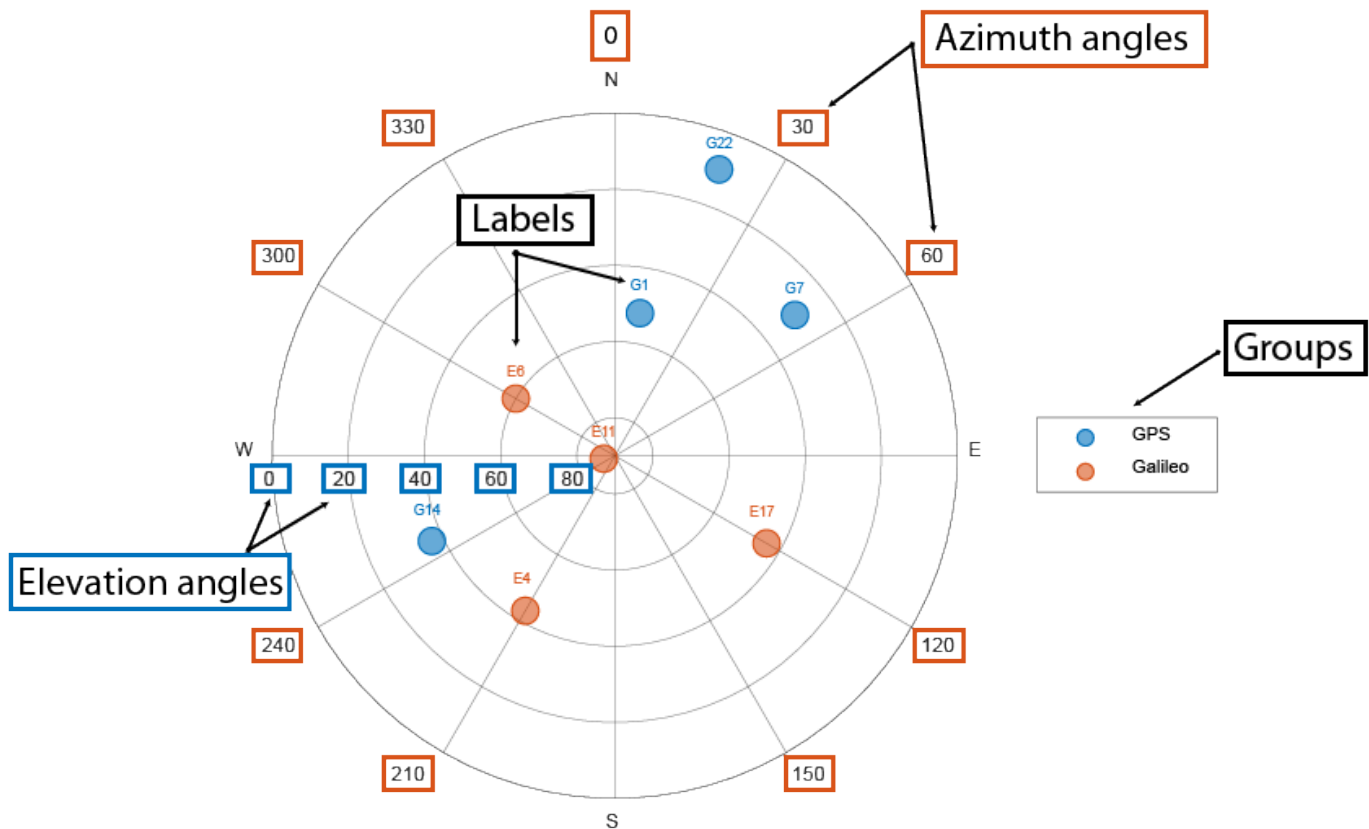
Global Navigation Satellite System Simulation Calculations: Access GNSS constellations, satellite look angles, pseudoranges, and receiver positions

Four new functions related to GNSS simulation have been added:

- `gnssconstellation` — Satellite locations at the specified time.
- `lookangles` — Satellite look angles from receiver and satellite positions.
- `pseudoranges` — Pseudoranges between the GNSS receiver and satellites.
- `receiverposition` — Estimate GNSS receiver position and velocity.

Sky Plot Visualization: Plot and label satellite positions in a polar plot

The skyplot function plots the azimuth and elevation angles for satellite positions. The function generates the plot as a SkyPlotChart Properties object.



These are the main elements of the figure:

- Azimuth axes — Specified by the `azdata` input argument, azimuth angle positions are measured clockwise from the North direction.
- Elevation axes — Specified by the `eldata` input argument, elevation angle positions are measured as the angle above the horizon line.
- Labels — Specified by the `labeldata` input argument as a string array in which each point in the `azdata` and `eldata` vectors represented by an element.
- Groups — Specified by the `GroupData` property, groups are a `categorical` array in which each satellite position represented by an element.

Comprehensive support for tuning inertial sensor filters

You can tune the parameters of these inertial sensor filter objects using their `tune` object function:

- `insfilterNonholonomic`
- `ahrs10filter`
- `insfilterMARG`
- `insfilterErrorState`

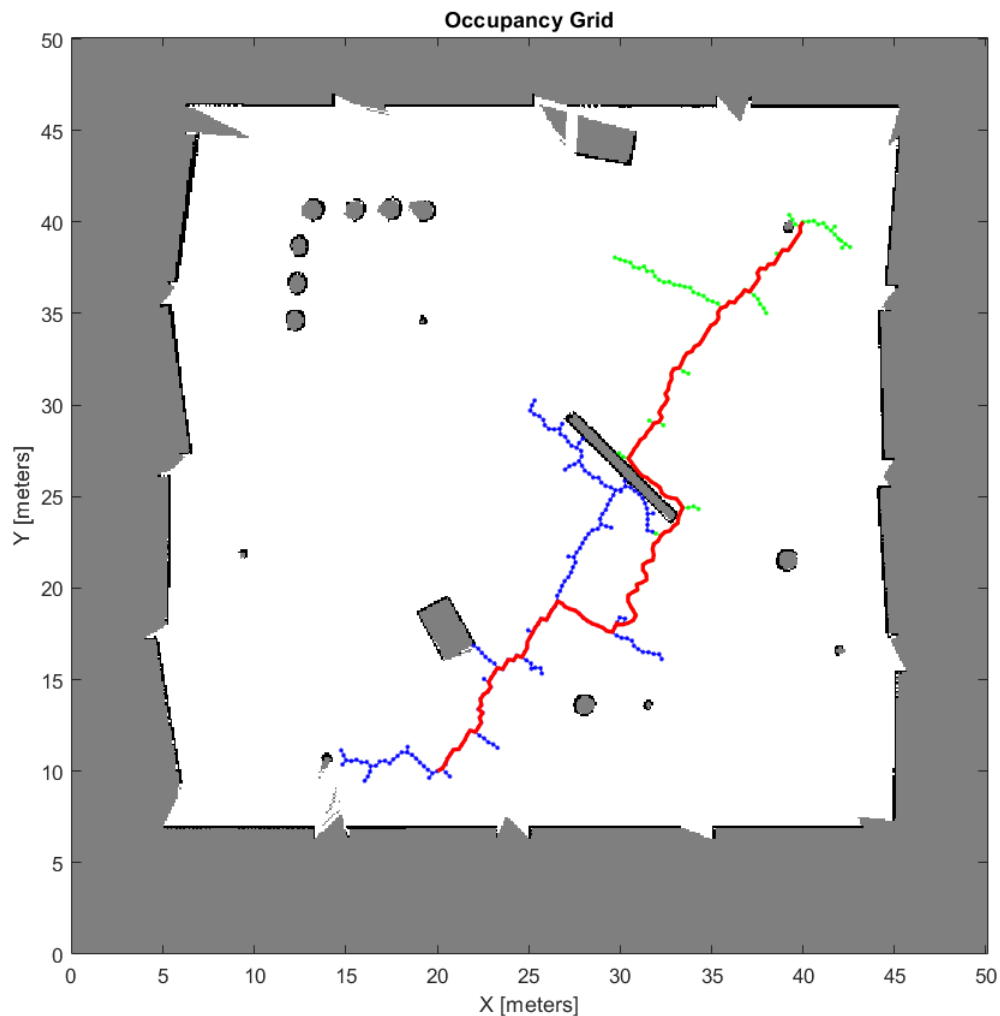
In the previous release, the other three inertial sensor filters (`imufilter`, `ahrsfilter`, and `insfilterAsync`) already supported the `tune` function.

The `tunerconfig` object, which is used to configure the tuning process, has three new properties:

- `Filter` — Class name of the fusion filter
- `FunctionTolerance` — Minimum change in cost to continue tuning
- `OutputFcn` — Output function to show tuning results. For example, you can use the `tunerPlotPose` function to visualize the truth data and state estimates after tuning.

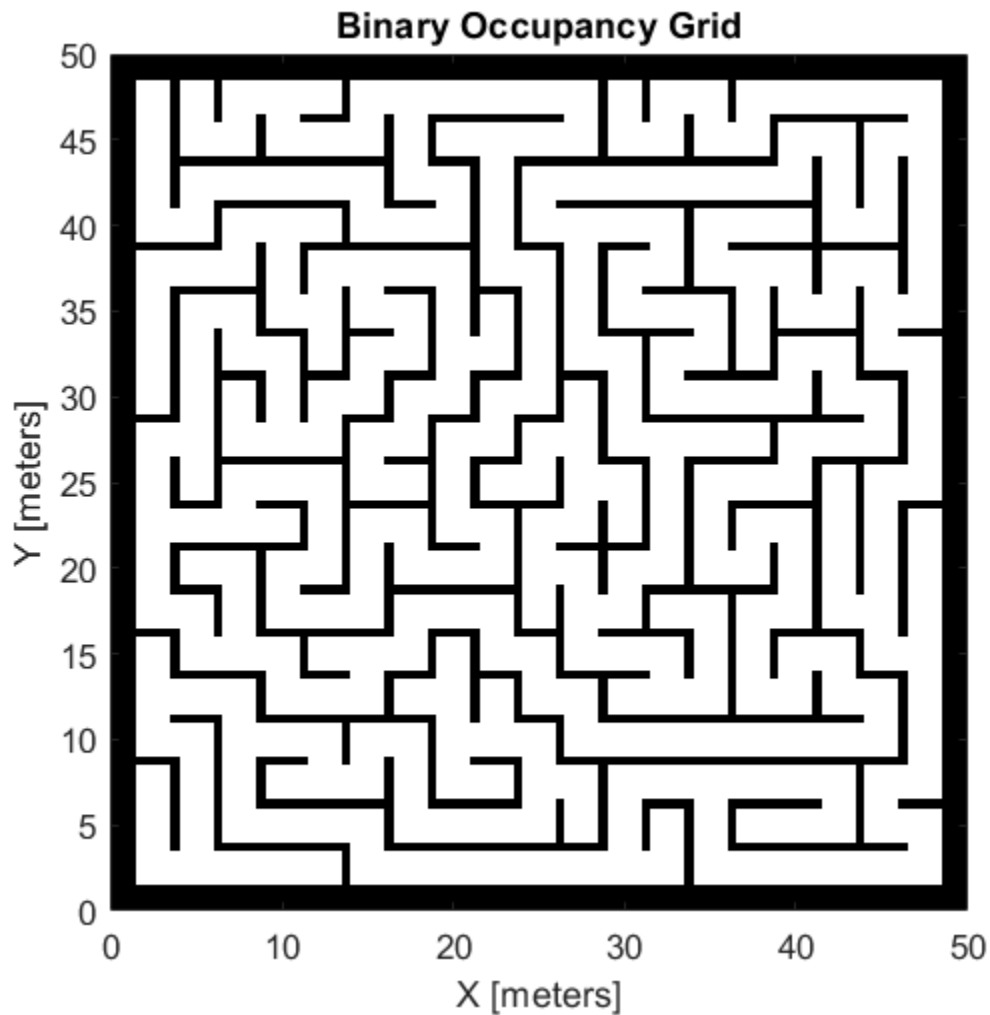
Bidirectional RRT Path Planner: Plan path using the bidirectional rapidly exploring random tree algorithm

The `plannerBiRRT` object plans an obstacle-free path from a start state to a goal state using the bidirectional RRT algorithm.



Generate Random Maze Maps for Planner Benchmarking

The `mapMaze` function generates a random maze map.



Transform Between Geodetic Coordinates and Local Cartesian Coordinates

Use these functions to transform between geodetic coordinates and local North-East-Down (NED) or East-North-Up (ENU) coordinates.

- `enu2lla` — Transform local ENU coordinates to geodetic coordinates.
- `ned2lla` — Transform local NED coordinates to geodetic coordinates.
- `lla2enu` — Transform geodetic coordinates to local ENU coordinates.
- `lla2ned` — Transform geodetic coordinates to local NED coordinates.

RRT* Path Planner Improvements: Returns PathCosts field in the solution information structure

The `plan` function of the `plannerRRTStar` object now returns the `PathCosts` field in the solution information structure.

Lidar SLAM Object Update: Specify scan registration method to use while adding scan

You can now specify which scan registration method to use when adding a scan to a `lidarSLAM` object by using the `ScanRegistrationMethod` property.

The `lidarSLAM` object now supports the `branchAndBound` (default) and `phaseCorrelation` scan registration algorithms. Using the `phaseCorrelation` algorithm requires an Image Processing Toolbox™ license.

optimizePoseGraph Function Update

The `optimizePoseGraph` function now accepts a pose graph representation specified as a `digraph` object whose edges are described by `affine3d` (Image Processing Toolbox) or `rigid3d` (Image Processing Toolbox) objects.

NMEA Parser System Object Update: Support for GSV sentences

You can now use the `nmeaParser` System object to extract National Marine Electronics Association (NMEA) data from GNSS Satellites in View (GSV) sentences that are compliant with the NMEA 0183 specification.

updateOccupancy Object Function Update

The `updateOccupancy` object function now accepts occupancy values as a matrix. The size of the matrix must be equal to the `GridSize` property of the `occupancyMap` object.

Use geodetic coordinates as inputs to gpsSensor

You can use geodetic coordinates as inputs to a `gpsSensor` System object. To enable this option, specify the `PositionInputFormat` property of the `gpsSensor` object as `'Geodetic'`.

insSensor provides more properties to specify its characteristics

The `insSensor` System object provides six new properties to model an inertial navigation system sensor:

- `MountingLocation` — Location of sensor on platform
- `AccelerationAccuracy` — Standard deviation of acceleration noise
- `AngularVelocityAccuracy` — Standard deviation of angular velocity noise
- `TimeInput` — Enable or disable input of simulation time

-
- `HasGNSSFix` — Enable or disable GNSS fix
 - `PositionErrorFactor` — Drift rate of position without GNSS fix

Variable-sized input support for timescope object

The `timescope` object allows you to visualize scalar or variable-sized input signals. If the signal is variable sized, the number of channels (columns) cannot change.

R2020b

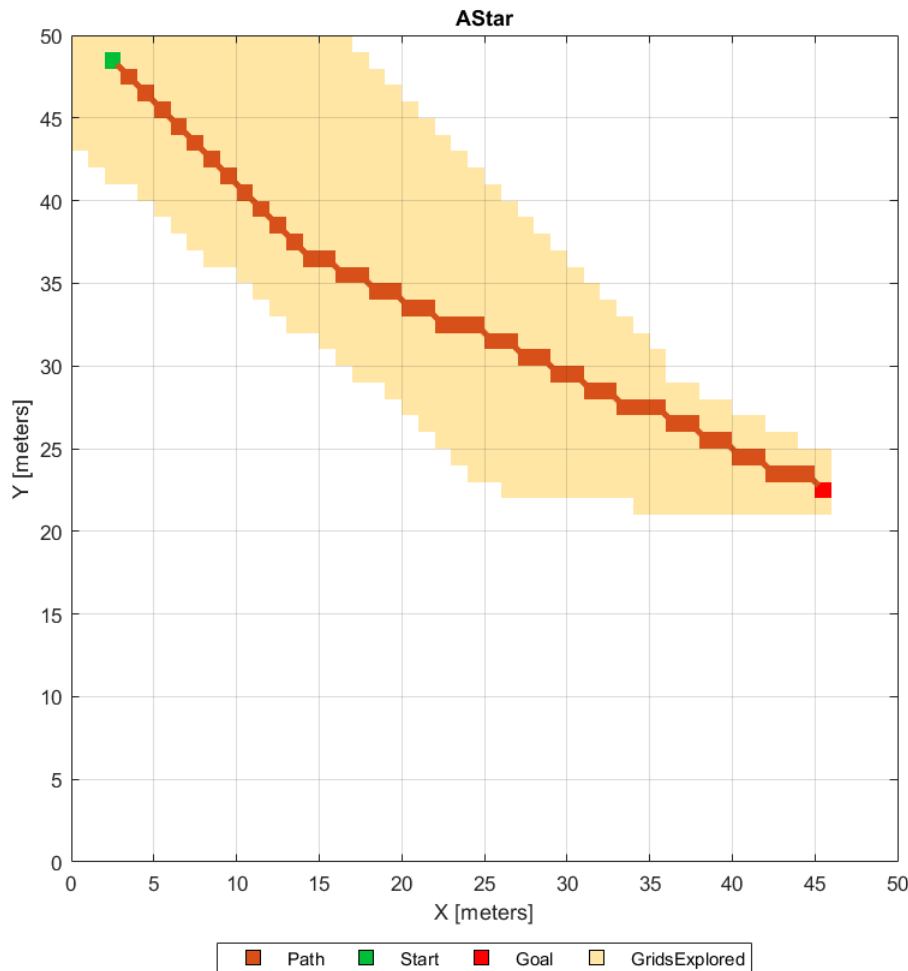
Version: 1.2

New Features

Bug Fixes

Grid-Based A* Path Planning: Plan path from start to goal location using the A* algorithm

Plan a path on a 2-D grid map using the `plannerAStarGrid` object.



Trajectory Optimal Frenet Utilities: More control in generating optimal trajectory in Frenet space

The `referencePathFrenet` object fits a smooth, piecewise continuous curve to the provided waypoints. Use the object functions to convert trajectories between global and Frenet coordinate systems, interpolate states along the path based on arc length, and query the closest point on the path from a global state.

The `trajectoryGeneratorFrenet` object generates trajectories between the initial and terminal states using fourth or fifth order polynomials. The trajectories are relative to a `referencePathFrenet` object. The `connect` function connects initial states to terminal states over a span of time.

For more details, see the Highway Trajectory Planning Using Frenet Reference Path example.

Dynamic Capsule-Based Obstacle: Model and simulate ego bodies and obstacles using collision primitive objects in environment

The `dynamicCapsuleList` and `dynamicCapsuleList3D` objects manage two lists of collision primitive objects: ego bodies and obstacles. Use the object functions to dynamically add, remove, and update the geometry and future poses of ego bodies and obstacles in the environment. To validate stored trajectories, use the `checkCollision` function, which checks the collisions between ego bodies and obstacles at each time step.

For more details, see the Highway Trajectory Planning Using Frenet Reference Path example.

SLAM Map Builder Update on Data Import: Import lidar scans and odometry data from MATLAB workspace

The **SLAM Map Builder** app now allows you to import lidar scans and odometry data from the MATLAB® workspace. To import data from the workspace, select **Import > Import from workspace**. Workspace import does not require a ROS Toolbox license.

Pose Graph Optimization Updates: Additional optimization functionality for pose graphs

The `poseGraph` and `poseGraph3D` objects now support the `edgeResidualErrors` function. This function computes edge residual errors for each edge in the pose graph given the current pose node estimates.

The `trimLoopClosures` function optimizes pose graphs by removing bad loop closure edges that would otherwise cause edge residual errors.

The `poseGraphSolverOptions` function creates solver options for pose graph optimization.

Wheel Encoder Sensor Models: Simulate wheel encoder sensor readings

Use provided sensor models to simulate wheel encoder sensor readings for various types of vehicles. Wheel encoder sensor models include:

- `wheelEncoderAckermann` object
- `wheelEncoderBicycle` object
- `wheelEncoderDifferentialDrive` object
- `wheelEncoderUnicycle` object

Wheel Encoder Odometry: Compute vehicle odometry using wheel encoder sensor readings

Use provided odometry models to compute vehicle odometry of various types of vehicles using wheel encoder sensor readings. Wheel encoder odometry models include:

- `wheelEncoderOdometryAckermann` object
- `wheelEncoderOdometryBicycle` object
- `wheelEncoderOdometryDifferentialDrive` object
- `wheelEncoderOdometryUnicycle` object

INS Sensor Model: Simulate inertial navigation and GPS readings

Use the `insSensor` object or INS block to simulate inertial navigation and GPS readings.

GNSS Sensor Model: Simulate GNSS receiver readings

Use the `gnssSensor` object to simulate Global Navigation Satellite System (GNSS) receiver readings.

Code Generation for 3-D Occupancy Map: Generate C/C++ code using `occupancyMap3D` object

You can now generate code when using the `occupancyMap3D` object.

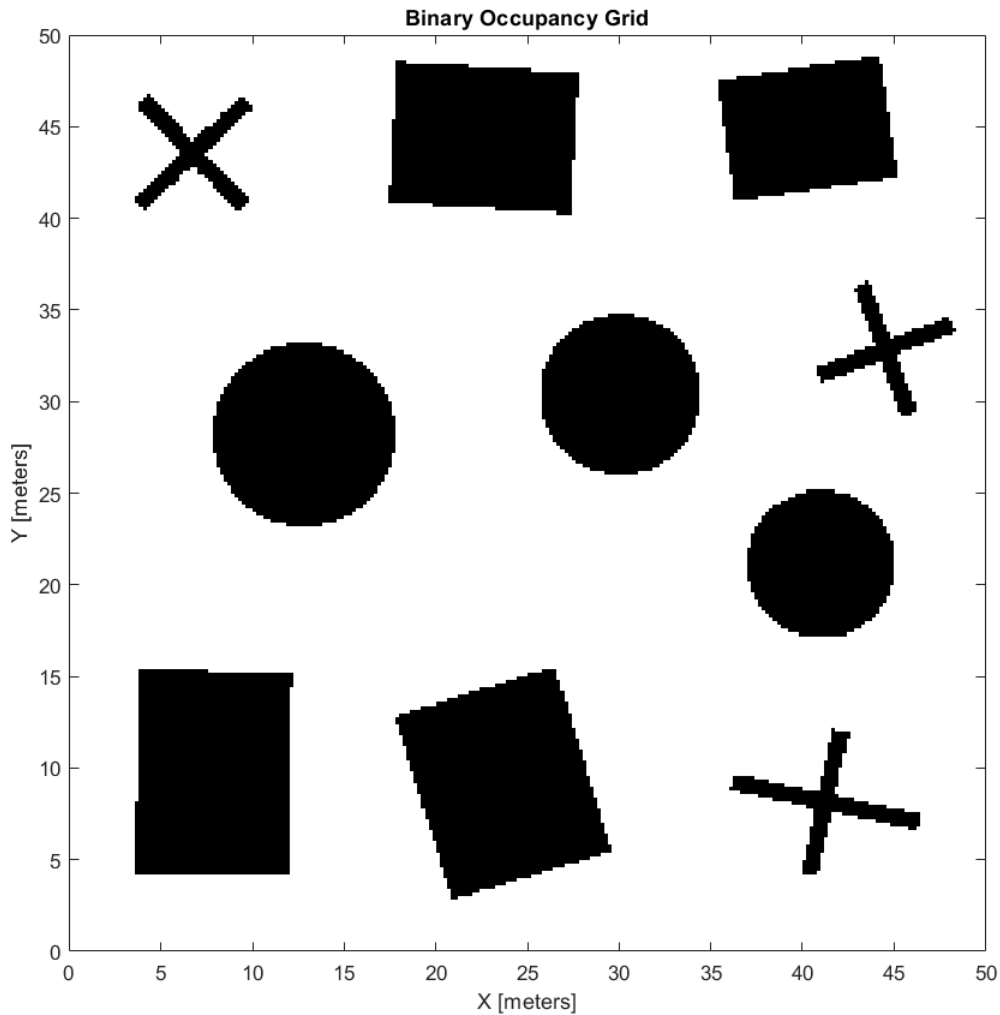
State Space and State Validator for 3-D Occupancy Map

Use the `stateSpaceSE3` object to store parameters and states in the 3-D state-space representation.

Use the `validatorOccupancyMap3D` object to validate SE3 states in a 3-D occupancy map.

Generate Random Grid Maps for Planner Benchmarking

Use the `mapClutter` function to generate a map with randomly scattered obstacles.



Adjust Inertial Sensor Fusion Performance Using Filter Tuner

Use the `tune` function and the `tunerconfig` object to adjust the properties of the `imufilter`, `ahrsfilter`, and `insfilterAsync` objects. Adjusting these properties can impact performance.

For more details, see the Automatic Tuning of the `insfilterAsync` Filter example.

GPS Device Object: Connect to GPS receiver from host computer

Use the `gpsdev` object to create connection to a GPS receiver connected to the host computer running Navigation Toolbox™. You can create the object either by specifying the serial port as an input argument or by using the `serialport` object from MATLAB.

After you create the `gpsdev` object, use these functions to perform further actions:

- `read` — Obtain GPS data like latitude, longitude and altitude (LLA), ground speed, course, dilution of precisions, and GPS receiver time, along with timestamp and overrun information.
- `flush` — Clear the software buffers and serial port buffers.
- `writeBytes` — Write raw data to configure the GPS receiver.

NMEA Parser Object: Parse data from standard NMEA sentences sent from GNSS receivers

Use the `nmeaParser` object to parse data from some of the standard NMEA (National Marine Electronics Association) sentences that are compliant with the NMEA 0183® specification.

The `nmeaParser` object parses data sent from GNSS receivers and identified by these NMEA message types: RMC, GGA, GSA, VTG, GLL, GST, ZDA, and HDT. The object outputs an array of structures corresponding to the data extracted from the requested NMEA message types.

Time Scope object: Bilevel measurements, triggers, and compiler support

The `timescope` object now includes support for:

- Bilevel measurements - Measure transitions, overshoots, undershoots, and cycles.
- Triggers - Set triggers to sync repeating signals and pause the display when events occur.
- MATLAB Compiler™ support - Use the `mcc` function to compile code for deployment.

New examples

This release contains several new examples:

- Wheel Encoder Error Sources
- Highway Trajectory Planning Using Frenet Reference Path
- GNSS Simulation Overview
- Automatic Tuning of the `insfilterAsync` Filter

R2020a

Version: 1.1

New Features

Bug Fixes

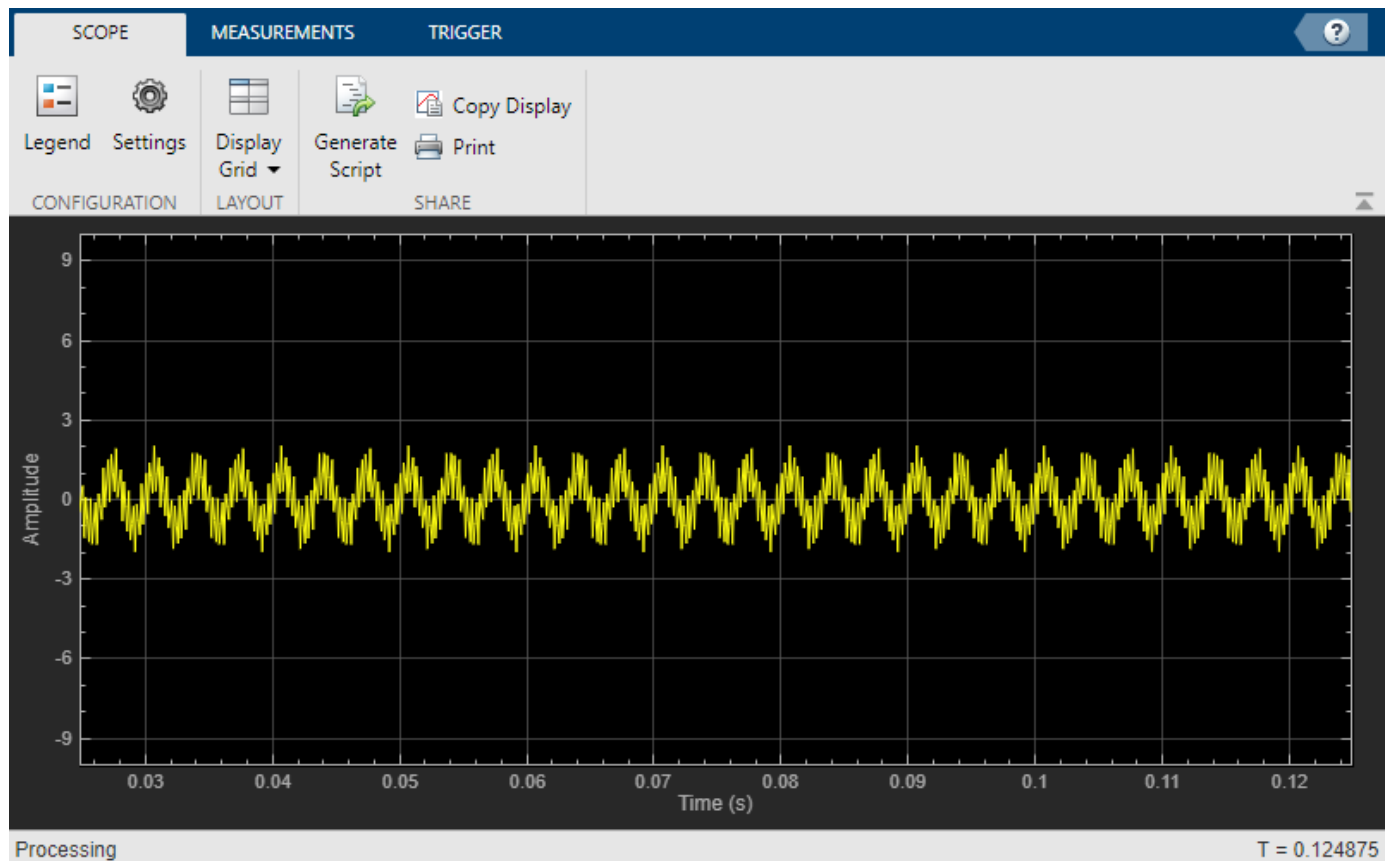
New Time Scope object: Visualize signals in the time domain

Use the `timescope` object to visualize real- and complex-valued floating-point and fixed-point signals in the time domain.

The Time Scope window has two toolstrip tabs:

Scope Tab

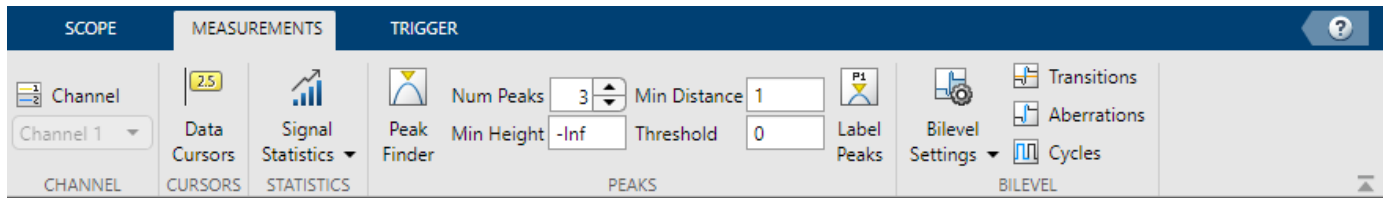
In the **Scope** tab, you can control the layout and configuration settings, and set the display settings of the Time Scope. You can also generate script to recreate your Time Scope with the same settings. When doing so, an editor window opens with the code required to recreate your `timescope` object.



Measurements Tab

In the **Measurements** tab, all measurements are made for a specified channel.





- **Data Cursors** -- Display the screen cursors.
- **Signal Statistics** -- Display the various statistics of the selected signal, such as maximum/minimum values, peak-to-peak values, mean, median, RMS.
- **Peak Finder** -- Display peak values for the selected signal.



Scale Axes

You can use the mouse to pan around the axes, and use the scroll button on your mouse to zoom in and out of the plot.

You can also use the buttons that appear when you hover over the plot window.

-  — Maximize the axes, hiding all labels and inseting the axes values.
-  — Zoom in on the plot.
-  — Pan around the axes.
-  — Autoscale the axes to fit the shown data.

For more details, see [Configure Time Scope MATLAB Object](#).

Scan Matching Using Line Features: Estimate pose and covariance based on line features in lidar scans

The `matchScansLine` function calculates a relative pose and estimated covariance between lidar scan readings based on estimated linear features.

Trajectory Optimization Improvements: Specify longitudinal segments, deviation offsets, and additional waypoint parameters

The `trajectoryOptimalFrenet` contains two new properties: `NumSegments` and `DeviationOffset`. Increasing `NumSegments` divides the longitudinal terminal states into multiple segments to calculate more dynamic trajectories, but increases computational complexity. `DeviationOffset` specifies an offset on the cost calculation for trajectories to bias the optimal trajectory in a specific direction that deviated from the reference path.

You can also calculate trajectories based on a velocity-keeping behavior by specifying `NaN` for the `Longitudinal` field of the `TerminalStates` property.

The `plan` function no longer errors when a feasible trajectory is not found. The function now returns an empty trajectory vector and an exit flag is included in the output arguments.

Path Metrics Improvements: Specify `validatorVehicleCostmap` as a state validator

The `pathmetrics` function now supports the `validatorVehicleCostmap` as the state validator input to the function.

Ray Intersections for 3-D Maps: Calculate ray intersections, import, and export with a 3-D occupancy map

The `occupancyMap3D` object now supports the `rayIntersection` function for calculating the intersection of rays with obstacles in the environment. You can also import and export occupancy maps as a `.bt` or `.ot` octomap file.

Code Generation for Monte Carlo Localization: Generate C/C++ code using the `monteCarloLocalization` object

You can now generate code when using the `monteCarloLocalization` object.

Code Generation for Sampling-Based Planners: Generate C/C++ code using the `plannerRRT`, `plannerRRTStar`, and `plannerHybridAStar` objects

You can now generate code when using the `plannerRRT`, `plannerRRTStar`, and `plannerHybridAStar` objects.

Code Generation for Trajectory Optimization: Generate C/C++ code using the `trajectoryOptimalFrenet` object

You can now generate code when using the `trajectoryOptimalFrenet` object.

Access residuals and residual covariance of insfilters and `ahrs10filter`

You can access the residuals and residual covariance information of insfilters (`insfilterMARG`, `insfilterAsync`, `insfilterErrorState`, and `insfilterNonholonomic`) and `ahrs10filter` through their object functions such as `fusegps`, `fusegyro`, `residual`, and `residualgps`.

Model inertial measurement unit using IMU Simulink block

Use the IMU Simulink block to model an inertial measurement unit (IMU) composed of accelerometer, gyroscope, and magnetometer sensors.

Estimate device orientation using AHRS Simulink block

Use the AHRS Simulink block to estimate the orientation of a device from its accelerometer, magnetometer, and gyroscope sensor readings.

Calculate angular velocity from quaternions

Use `angvel` to calculate angular velocity from an array of quaternions.

Transform position and velocity between two frames to motion quantities in a third frame

Use `transformMotion` to transform position and velocity between two coordinate frames to motion quantities in a third coordinate frame.

R2019b

Version: 1.0

New Features

Simultaneous Localization and Mapping (SLAM): Create 2-D and 3-D occupancy maps using SLAM algorithm and lidar scan data

Use the SLAM algorithm to tune parameters for scan matching and loop-closure detection. The `LidarSLAM` object takes lidar scan data and builds a map as your vehicle moves through it. The algorithm generates a `poseGraph` and continuously optimizes edge-constraints based on detected loop closures. As more loop closures are detected, you can continuously build a map of your environment and adjust for odometry drift.

For an example using 2-D lidar scans, see [Implement Online Simultaneous Localization And Mapping \(SLAM\) with Lidar Scans](#).

For an example using 3-D lidar point clouds, see [Perform SLAM Using 3-D Lidar Point Clouds](#).

For more information, see [SLAM](#).

SLAM Map Builder App: Interactively modify loop closures and adjust overall map using SLAM algorithm

Use the **SLAM Map Builder** app to load and filter lidar scans and estimated poses from a log file or data in the workspace. Tune and run the SLAM algorithm to automatically build the map. Pause at any time to modify relative poses between scans. Modify or delete loop closures from the pose graph to improve the overall map. After you are done with the entire data set, output the map as an occupancy grid to use with path planning or other navigation algorithms.

Pose Estimation: Accurately estimate vehicle poses using IMU and GPS sensors and Monte Carlo Localization

Use localization and pose estimation algorithms to orient your vehicle in your environment. Sensor pose estimation uses filters to improve and combine sensor readings for IMU, GPS, and other sensors. Localization algorithms, like Monte Carlo localization and scan matching, estimate your pose in a known map using range sensor or lidar readings. Pose graphs track your estimated poses and can be optimized based on edge constraints and loop closures.

For more information, see [Localization and Pose Estimation](#)

Customizable Sampling-Based Path Planners: Plan a path from start to goal locations using RRT and RRT* algorithms

Plan paths through a 2-D environment using provided path planning algorithms:

- `plannerRRT`
- `plannerRRTStar`
- `plannerHybridAStar`

Specify parameters for provided 2-D state-space representations:

- `stateSpaceSE2`
- `stateSpaceDubins`

-
- `stateSpaceReedsShepp`

Validate your planned paths using occupancy maps or vehicle cost maps:

- `validatorOccupancyMap`
- `validatorVehicleCostmap`

Write your own custom state space or state validator using class interfaces:

- `nav.StateSpace`
- `nav.StateValidator`

Path-Planning Metrics: Use metrics to check and compare the output of path planners

Calculate path metrics to evaluate planned paths using the `pathmetrics` object. Check the clearance and smoothness based on your path constraints.

Sensor Models: Use simulated models for IMU, GPS, and range sensors

Perform sensor modeling and simulation for accelerometers, magnetometers, gyroscopes, altimeters, GPS, IMU, and range sensors. Analyze sensor readings, sensor noise, environmental conditions, and other configuration parameters. Generate trajectories to emulate these sensors traveling through a world, and calibrate the performance of your sensors.

Sensor models include:

- `gpsSensor`
- `imuSensor`
- `rangeSensor`

For other sensors and more information, see [Sensor Models](#).

Trajectory and Waypoint Following Algorithms: Use built-in algorithms to generate trajectories and control commands for robots

Use the `waypointTrajectory` and `kinematicTrajectory` objects to generate trajectories for sensors or vehicles and control commands to send to your vehicle

