

**SimBiology<sup>®</sup>**

Reference



**MATLAB<sup>®</sup>**

R2018b



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*SimBiology*<sup>®</sup> Reference

© COPYRIGHT 2005–2018 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](http://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](http://www.mathworks.com/patents) for more information.

## Revision History

September 2005	Online only	New for Version 1.0 (Release 14SP3+)
March 2006	Online only	Updated for Version 1.0.1 (Release 2006a)
May 2006	Online only	Updated for Version 2.0 (Release 2006a+)
September 2006	Online only	Updated for Version 2.0.1 (Release 2006b)
March 2007	Online only	Rereleased for Version 2.1.1 (Release 2007a)
September 2007	Online only	Rereleased for Version 2.1.2 (Release 2007b)
October 2007	Online only	Updated for Version 2.2 (Release 2007b+)
March 2008	Online only	Updated for Version 2.3 (Release 2008a)
October 2008	Online only	Updated for Version 2.4 (Release 2008b)
March 2009	Online only	Updated for Version 3.0 (Release 2009a)
September 2009	Online only	Updated for Version 3.1 (Release 2009b)
March 2010	Online only	Updated for Version 3.2 (Release 2010a)
September 2010	Online only	Updated for Version 3.3 (Release 2010b)
April 2011	Online only	Updated for Version 3.4 (Release 2011a)
September 2011	Online only	Updated for Version 4.0 (Release 2011b)
March 2012	Online only	Updated for Version 4.1 (Release 2012a)
September 2012	Online only	Updated for Version 4.2 (Release 2012b)
March 2013	Online only	Updated for Version 4.3 (Release 2013a)
September 2013	Online only	Updated for Version 4.3.1 (Release 2013b)
March 2014	Online only	Updated for Version 5.0 (Release 2014a)
October 2014	Online only	Updated for Version 5.1 (Release 2014b)
March 2015	Online only	Updated for Version 5.2 (Release 2015a)
September 2015	Online only	Updated for Version 5.3 (Release 2015b)
March 2016	Online only	Updated for Version 5.4 (Release 2016a)
September 2016	Online only	Updated for Version 5.5 (Release 2016b)
March 2017	Online only	Updated for Version 5.6 (Release 2017a)
September 2017	Online only	Updated for Version 5.7 (Release 2017b)
March 2018	Online only	Updated for Version 5.8 (Release 2018a)
September 2018	Online only	Updated for Version 5.8.1 (Release 2018b)



**1** | Functions – Alphabetical List

**2** | Methods – Alphabetical List

**3** | Properties – Alphabetical List



# Functions — Alphabetical List

---

## sbioabstractkineticlaw

Create kinetic law definition

### Syntax

```
abstkineticlawObj = sbioabstractkineticlaw('Name')  
abstkineticlawObj = sbioabstractkineticlaw('Name','Expression')
```

```
abstkineticlawObj = sbioabstractkineticlaw(...'PropertyName',  
PropertyValue...)
```

### Arguments

<i>Name</i>	Enter a name for the kinetic law definition. <i>Name</i> can be a character vector or string. It must be unique in the user-defined kinetic law library. <i>Name</i> is referenced by <i>kineticlawObj</i> .
<i>Expression</i>	The mathematical expression that defines the kinetic law.

### Description

*abstkineticlawObj* = `sbioabstractkineticlaw('Name')` creates an abstract kinetic law object, with the name *Name* and returns it to *abstkineticlawObj*. Use the abstract kinetic law object to specify a *kinetic law definition*.

The *kinetic law definition* provides a mechanism for applying a specific rate law to multiple reactions. It acts as a mapping template for the reaction rate. The kinetic law definition defines a reaction rate expression, which is shown in the property `Expression`, and the species and parameter variables used in the expression. The species variables are defined in the `SpeciesVariables` on page 3-217 property, and the parameter variables are defined in the `ParameterVariables` on page 3-158 property of the abstract kinetic law object.



To use the kinetic law definition, you must add it to the user-defined library with the `sbioaddtolibrary` function. To retrieve the kinetic law definitions from the user-defined library, first create a root object using `sbiroot`, then use the command `get(rootObj.UserDefinedLibrary, 'KineticLaws')`.

`abstkineticlawObj = sbioabstractkineticlaw('Name', 'Expression')` constructs a SimBiology abstract kinetic law object, `abstkineticlawObj` with the name `'Name'` and with the expression `'Expression'` and returns it to `abstkineticlawObj`.

`abstkineticlawObj = sbioabstractkineticlaw(...'PropertyName', PropertyValue...)` defines optional properties. The name-value pairs can be in any format supported by the function `set`.

Additional `abstkineticlawObj` properties can be viewed with the `get` command. `abstkineticlawObj` properties can be modified with the `set` command.

---

**Note** If you use the `sbioabstractkineticlaw` constructor function to create an object containing a reaction rate expression that is not continuous and differentiable, see “Using Events to Address Discontinuities in Rule and Reaction Rate Expressions” before simulating your model.

---

## Method Summary

<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>findUsages</code> (AbstractKineticLaw)	Find out how an AbstractKineticLaw object is used
<code>get</code> (any object)	Get object properties
<code>set</code> (any object)	Set object properties

## Property Summary

Expression (AbstractKineticLaw, KineticLaw)	Expression to determine reaction rate equation
Name	Specify name of object
Notes	HTML text describing SimBiology object
ParameterVariables	Parameters in kinetic law definition
Parent	Indicate parent object
SpeciesVariables	Species in abstract kinetic law
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

- 1 Create a kinetic law definition.

```
abstkineticlawObj = sbioabstractkineticlaw('ex_myLaw1', '(k1*s)/(k2+k1+s)');
```

- 2 Assign the parameter and species variables in the expression.

```
set (abstkineticlawObj, 'SpeciesVariables', {'s'});
set (abstkineticlawObj, 'ParameterVariables', {'k1', 'k2'});
```

- 3 Add the new kinetic law definition to the user-defined library.

```
sbioaddtolibrary(abstkineticlawObj);
```

`sbioaddtolibrary` adds the kinetic law definition to the user-defined library. You can verify this using `sbiowhos`.

```
sbiowhos -kineticlaw -userdefined
```

```
SimBiology Abstract Kinetic Law Array
```

Index:	Library:	Name:	Expression:
1	UserDefined	ex_myLaw1	(k1*s)/(k2+k1+s)

- 4 Use the new kinetic law definition when defining a reaction's kinetic law.

```
modelObj = sbiomodel('cell');  
reactionObj = addreaction(modelObj, 'A + B <-> B + C');  
kineticlawObj = addkineticlaw(reactionObj, 'ex_myLaw1');
```

---

**Note** Remember to specify the `SpeciesVariableNames` and the `ParameterVariableNames` in `kineticlawObj` to fully define the `ReactionRate` of the reaction.

---

## See Also

[addkineticlaw](#) | [addparameter](#) | [addreaction](#) | [sbiomodel](#)

**Introduced in R2006a**

## sbioaccelerate

Prepare model object for accelerated simulations

### Syntax

```
sbioaccelerate(modelObj)
```

```
sbioaccelerate(modelObj,csObj)
```

```
sbioaccelerate(modelObj,dvObj)
```

```
sbioaccelerate(modelObj,csObj,dvObj)
```

```
sbioaccelerate(modelObj,csObj,variantObj,doseObj)
```

### Description

`sbioaccelerate(modelObj)` prepares a model object for an accelerated simulation using its active configuration set (`configset`), any active variants and active doses. A SimBiology model can contain multiple configsets with only one being active at any given time. The active configset contains the settings to use in model preparation for acceleration.

For accelerated simulations, use `sbioaccelerate` before running `sbiosimulate`. You must use the same model and configset for both functions.

Rerun `sbioaccelerate`, before calling `sbiosimulate`, if you modify this model, such as changing reactions or adding events. However, there are exceptions. For details, see “When to Rerun Acceleration”.

---

**Note** If you are using a `SimFunction` object for simulations, it automatically accelerates the model on its first function evaluation. Hence it is not necessary to run `sbioaccelerate` beforehand.

---

---

**Prerequisites** To prepare your models for accelerated simulations, install and set up a supported compiler. For details, see “Prerequisites for Accelerating Simulations and Analyses”.

---

`sbioaccelerate(modelObj, csObj)` uses the specified configset object `csObj` and any active variants and active doses. Any other configsets are ignored. If you set `csObj` to empty `[]`, the function uses the active configset.

`sbioaccelerate(modelObj, dvObj)` uses doses or variants specified by `dvObj` and the active configset. `dvObj` can be one of the following:

- Variant object
- ScheduleDose object
- RepeatDose object
- array of doses or variants

If you set `dvObj` to empty `[]`, the function uses the active configset, active variants, and active doses.

If you specify `dvObj` as variants, the function uses the specified variants and active doses. Any other variants are ignored.

If you specify `dvObj` as doses, the function uses the specified doses and active variants. Any other doses are ignored.

Currently, a particular dose object can only be accelerated with a single model. You cannot use the same dose object for multiple models to be accelerated. You must create a new copy of the dose for each model.

`sbioaccelerate(modelObj, csObj, dvObj)` uses a configset object `csObj` and doses or variants specified by `dvObj`.

If you set `csObj` to `[]`, then the function uses the active configset object.

If you set `dvObj` to `[]`, then the function uses no variants, but uses active doses.

If you set `dvObj` to variants, the function uses the specified variants and active doses. Any other variants are ignored.

If you set `dvObj` to doses, the function uses the specified doses and active variants. Any other doses are ignored.

`sbioaccelerate(modelObj, csObj, variantObj, doseObj)` uses a configset object `csObj`, variant object or variant array specified by `variantObj` and dose object or dose array specified by `doseObj`. Any other configset, doses, and variants are ignored.

If you set `csObj` to `[]`, then the function uses the active configset object.

If you set `variantObj` to `[]`, then the function uses no variants.

If you set `doseObj` to `[]`, then the function uses no doses.

## Examples

### Prepare a Model for Accelerated Simulation

Load a SimBiology project, named `lotka`, that contains a model `m1`.

```
sbioloadproject('lotka','m1')
```

Prepare the model for accelerated simulation.

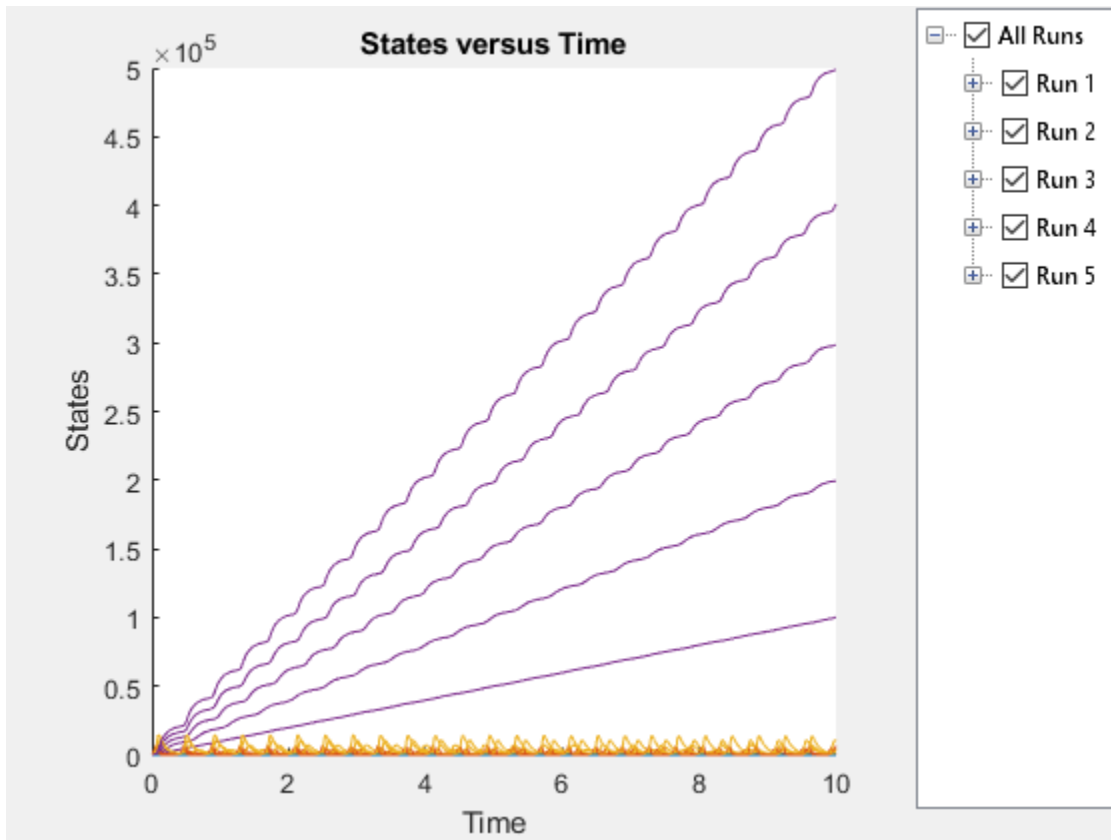
```
sbioaccelerate(m1);
```

Simulate the model using different initial amounts of species `x`.

```
x = sbioselect(m1, 'type', 'species', 'name', 'x');  
for i=1:5  
    x.initialAmount = i;  
    sd(i) = sbiosimulate(m1);  
end
```

Plot the results.

```
sbioplot(sd);
```



### Accelerate Simulation Using a User-Defined Configset Object

Load a sample SimBiology project.

```
sbioloadproject radiodecay.sbproj
```

Add a new configuration set using a different stop time of 15 seconds.

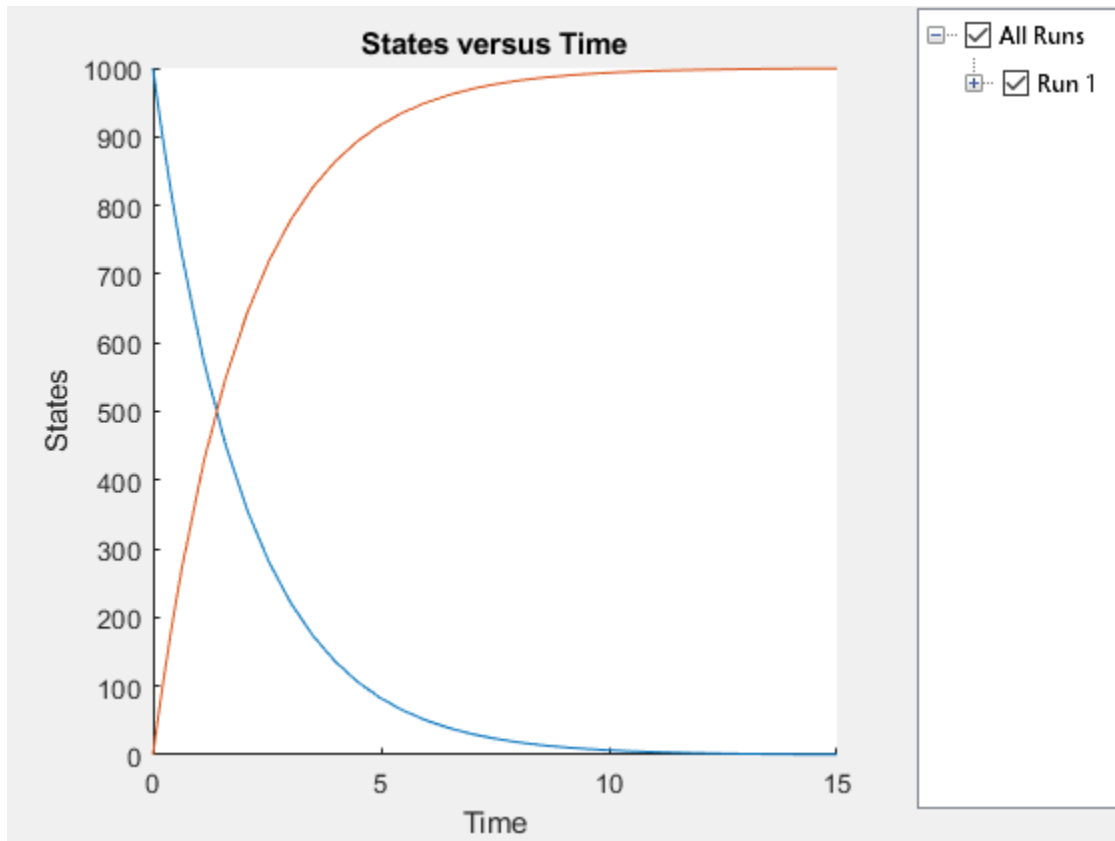
```
csObj = addconfigset(m1, 'newStopTimeConfigSet');
csObj.StopTime = 15;
```

Prepare the model for accelerated simulation using the new configset object.

```
sbioaccelerate(m1,cs0bj);
```

Simulate the model using the same configset object.

```
sim = sbiosimulate(m1,cs0bj);  
sbioplot(sim);
```



## Accelerate Simulation With Array of Doses

Load a sample SimBiology project.

```
sbioloadproject radiodecay.sbproj
```



Increase the amount of species x by 100 molecules at 2 and 4 seconds by adding a schedule dose.

```
dObj1 = adddose(m1, 'd1', 'schedule');
dObj1.Amount = 100;
dObj1.AmountUnits = 'molecule';
dObj1.TimeUnits = 'second';
dObj1.Time = 2;
dObj1.TargetName = 'unnamed.x';

dObj2 = adddose(m1, 'd2', 'schedule');
dObj2.Amount = 100;
dObj2.AmountUnits = 'molecule';
dObj2.TimeUnits = 'second';
dObj2.Time = 4;
dObj2.TargetName = 'unnamed.x';
```

Prepare the model for accelerated simulation using the array of both doses.

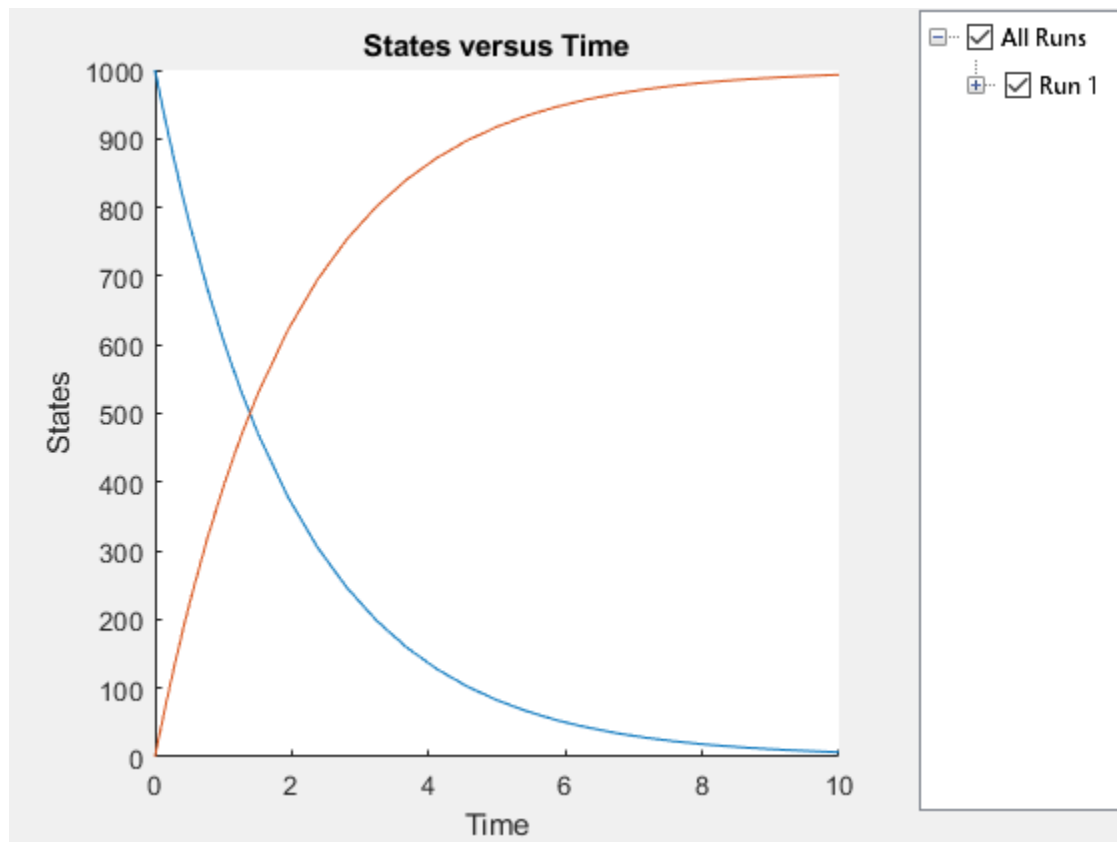
```
sbioaccelerate(m1, [dObj1, dObj2]);
```

Simulate the model using no dose or any subset of the dose array without having to rerun sbioaccelerate.

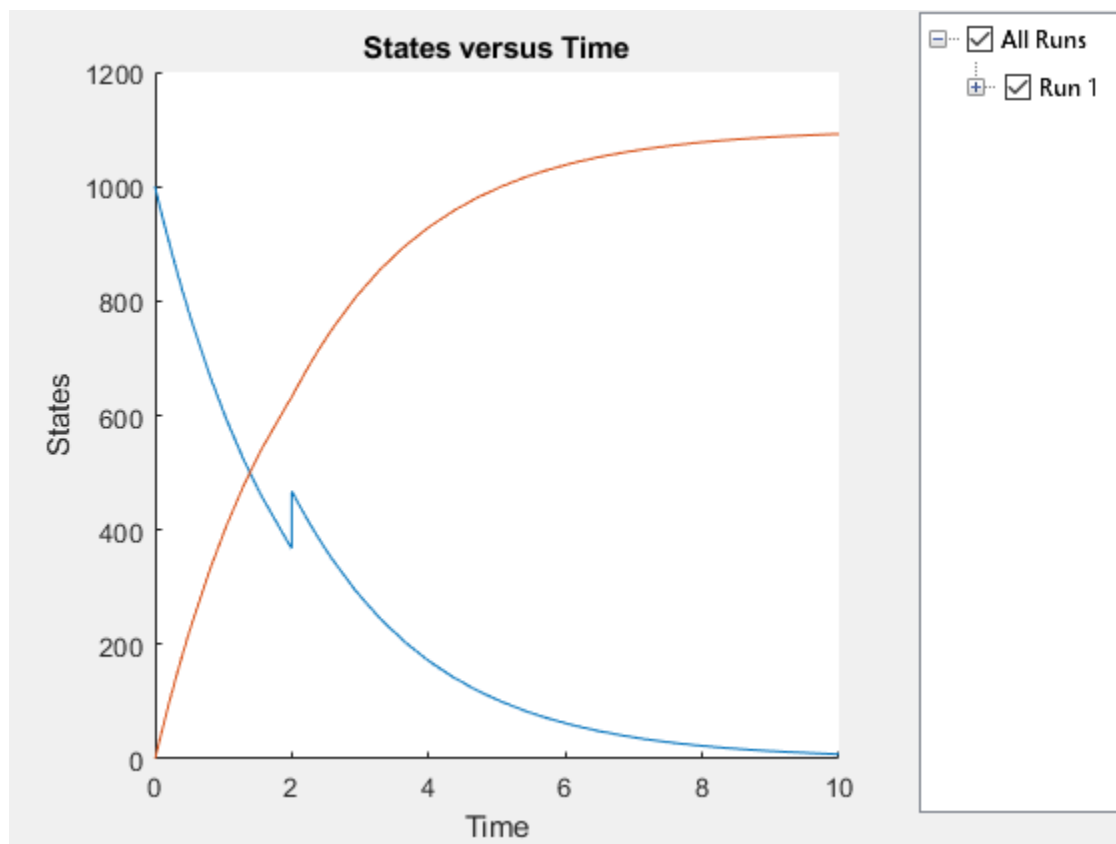
```
sim1 = sbiosimulate(m1);
sim2 = sbiosimulate(m1, dObj1);
sim3 = sbiosimulate(m1, dObj2);
sim4 = sbiosimulate(m1, [dObj1, dObj2]);
```

Plot the results.

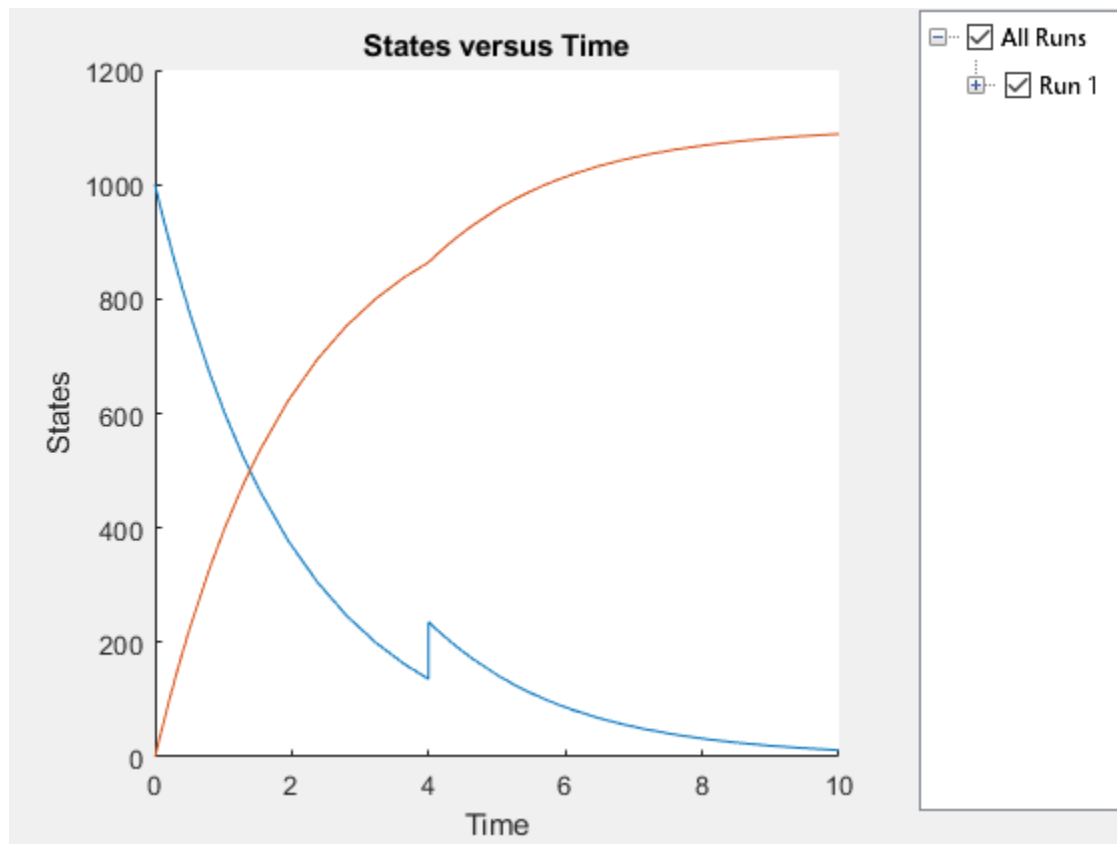
```
sbioplot(sim1);
```



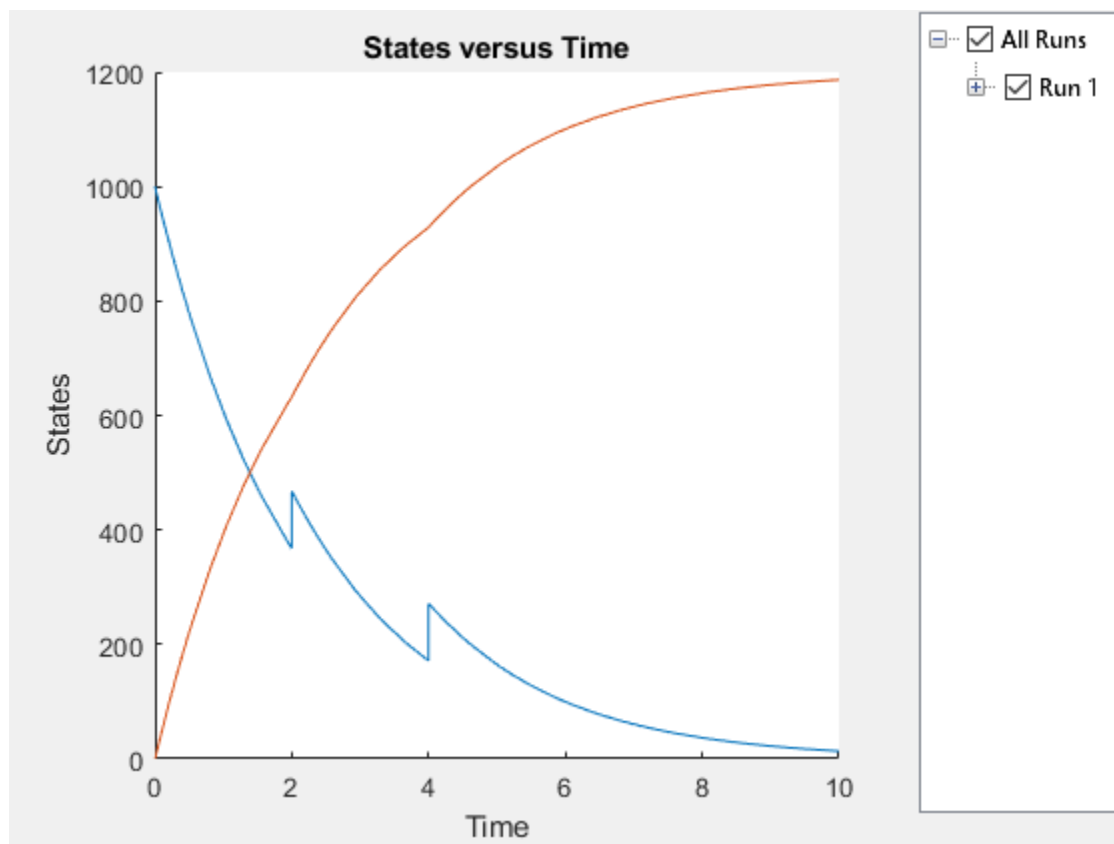
```
sbioplot(sim2);
```



```
sbioplot(sim3);
```



```
sbioplot(sim4);
```



### Accelerate Simulation Using Configset and Dose Objects

Load a sample SimBiology project.

```
sbioloadproject radiodecay.sbproj
```

Get the default configuration set from the model.

```
defaultConfigSet = getConfigset(m1, 'default');
```

Increase the amount of species x by 100 molecules at 2 seconds by adding a schedule dose.

```
dObj = adddose(m1, 'd1', 'schedule');  
dObj.Amount = 100;  
dObj.AmountUnits = 'molecule';  
dObj.TimeUnits = 'second';  
dObj.Time = 2;  
dObj.TargetName = 'unnamed.x';
```

Prepare the model for accelerated simulation using the default configset object and added dose object.

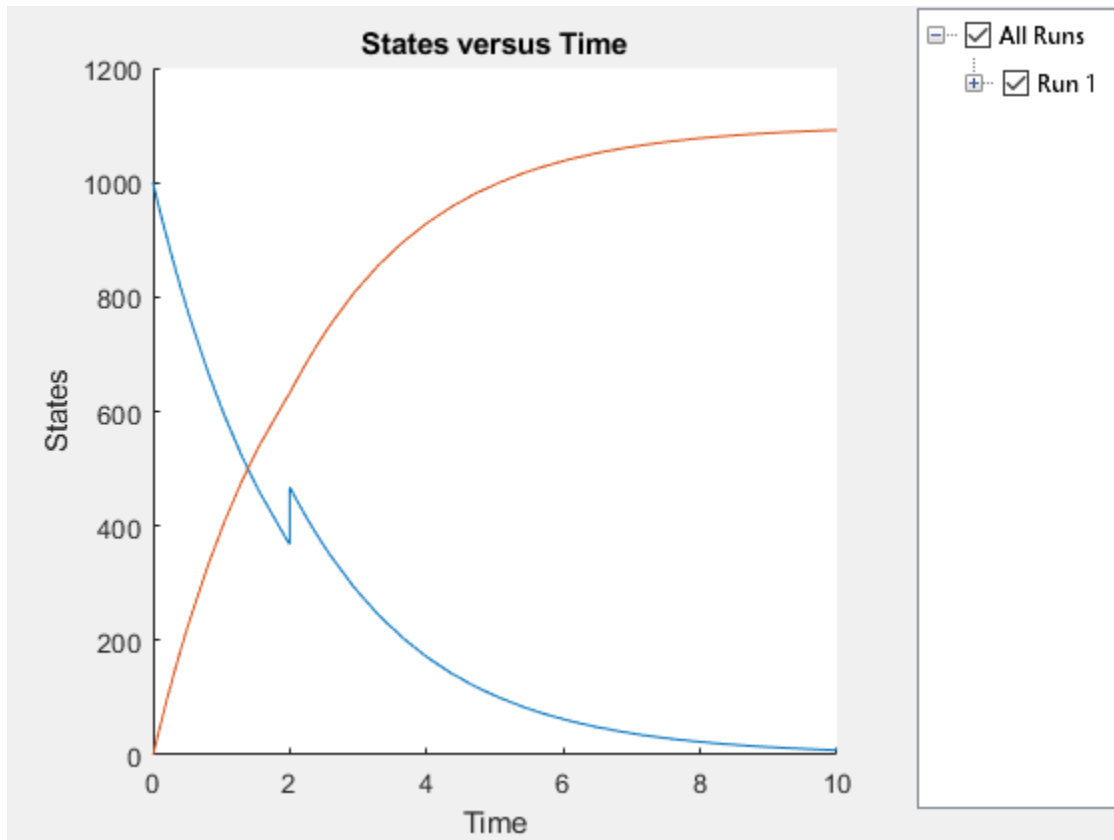
```
sbioaccelerate(m1, defaultConfigSet, dObj);
```

Simulate the model using the same configset and dose objects.

```
sim = sbiosimulate(m1, defaultConfigSet, dObj);
```

Plot the result.

```
sbioplot(sim);
```



### Accelerate Simulation Using Configset, Dose, and Variant Objects

Load a sample SimBiology project.

```
sbioloadproject radiodecay.sbproj
```

Add a new configuration set using a different stop time of 15 seconds.

```
csObj = m1.addconfigset('newStopTimeConfigSet');  
csObj.StopTime = 15;
```

Increase the amount of species *x* by 100 molecules at 2 seconds by adding a schedule dose.

```
dObj = adddose(m1, 'd1', 'schedule');  
dObj.Amount = 100;  
dObj.AmountUnits = 'molecule';  
dObj.TimeUnits = 'second';  
dObj.Time = 2;  
dObj.TargetName = 'unnamed.x';
```

Add a variant of species *x* using a different initial amount of 500 molecules.

```
vObj = addvariant(m1, 'v1');  
addcontent(vObj, {'species', 'x', 'InitialAmount', 500});
```

Prepare the model for accelerated simulation using the configset, dose, and variant objects. In this case, the third argument of `sbioaccelerate` must be the variant object.

```
sbioaccelerate(m1, csObj, vObj, dObj);
```

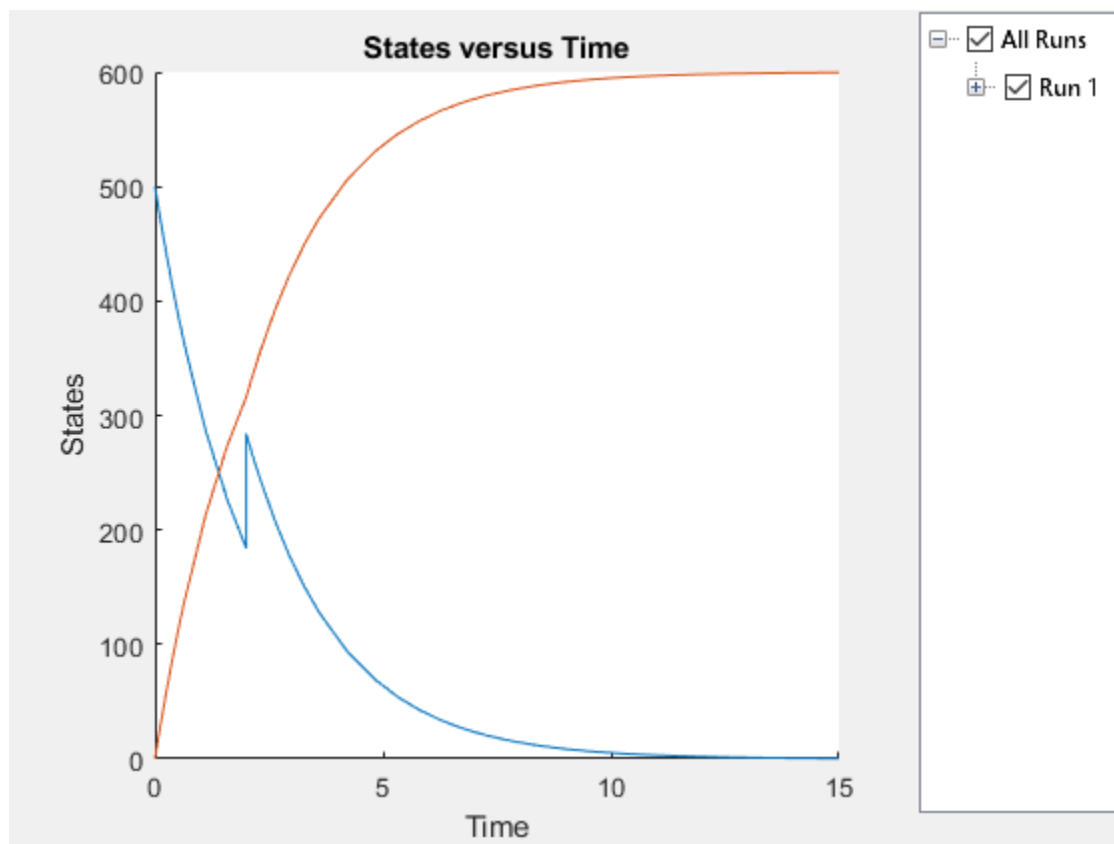
Simulate the model using the same configset, variant, and dose objects.

```
sim = sbiosimulate(m1, csObj, vObj, dObj);
```

Plot the result.

```
sbioplot(sim);
```





## Input Arguments

### **modelObj** — SimBiology model

SimBiology model object

SimBiology model, specified as a SimBiology model object. The model minimally needs one reaction or rate rule to be accelerated for simulations.

### **csObj** — Configuration set object

configset object | []

Configuration set object, specified as a `configset` object that stores simulation-specific information. When you specify `csObj` as [], `sbioaccelerate` uses the currently active `configset`.

### **dvObj — Dose or variant object**

dose object or array of dose objects | variant object or array of variant objects | []

Dose or variant object, specified as one of the following: `scheduleDose` object, `repeatDose` object, array of dose objects, `variant` object, or array of variant objects.

- Use [] when you want to explicitly exclude any variant objects from the `sbioaccelerate` function.
- When `dvObj` is a dose object, `sbioaccelerate` uses the specified dose object as well as any active variant objects if available. When you accelerate the model using an array of dose objects, you can simulate the model using any subset of the dose objects from the array.
- When `dvObj` is a variant object, `sbioaccelerate` uses the specified variant object as well as any active dose objects if available. You can use any or no variant input arguments when running `sbioaccelerate`.

### **variantObj — Variant object**

variant object or array of variant objects | []

Variant object, specified as a `variant` object or array of variant objects. Use [] when you want to explicitly exclude any variant object from `sbioaccelerate`.

### **doseObj — Dose object**

dose object or array of dose objects | []

Dose object, specified as a `scheduleDose` object, `repeatDose` object, or array of dose objects. A dose object defines additions that are made to species amounts or parameter values. Use [] when you want to explicitly exclude any dose objects from `sbioaccelerate`.

---

**Note** If you pass in an array of doses to `sbioaccelerate`, you can simulate the model using any subset of doses and do not need to run acceleration again.

---

## See Also

SimFunction object | sbiosimulate

## Topics

“Accelerating Model Simulations and Analyses”

**Introduced in R2010a**

## sbioaddtolibrary

Add to user-defined library

### Syntax

```
sbioaddtolibrary (abstkineticlawObj)  
sbioaddtolibrary (unitObj)  
sbioaddtolibrary (unitprefixObj)
```

### Arguments

<i>abstkineticlawObj</i>	Specify the abstract kinetic law object that holds the kinetic law definition. The Name of the kinetic law must be unique in the user-defined kinetic law library. Name is referenced by <i>kineticlawObj</i> . For more information about creating <i>kineticlawObj</i> , see <i>sbioabstractkineticlaw</i> .
<i>unitObj</i>	Specify the user-defined unit to add to the library. For more information about creating <i>unitObj</i> , see <i>sbiounit</i> .
<i>unitprefixObj</i>	Specify the user-defined unit prefix to add to the library. For more information about creating <i>unitprefixObj</i> , see <i>sbiounitprefix</i> .

### Description

The function *sbioaddtolibrary* adds kinetic law definitions, units, and unit prefixes to the user-defined library.

*sbioaddtolibrary* (*abstkineticlawObj*) adds the abstract kinetic law object (*abstkineticlawObj*) to the user-defined library.

*sbioaddtolibrary* (*unitObj*) adds the user-defined unit (*unitObj*) to the user-defined library.

`sbioaddtolibrary (unitprefixObj)` adds the user-defined unit prefix (`unitprefixObj`) to the user-defined library.

The `sbioaddtolibrary` function adds any kinetic law definition, unit, or unit prefix to the root object's `UserDefinedLibrary` property. These library components become available automatically in future MATLAB® sessions.

Use the kinetic law definitions in the built-in and user-defined library to construct a kinetic law object with the method `addkineticlaw`.

To get a component of the built-in and user-defined libraries, use the commands `get(sbioroot, 'BuiltInLibrary')` and `(get(sbioroot, 'UserDefinedLibrary'))`.

To remove the library component from the user-defined library, use the function `sbioremovefromlibrary`. You cannot remove a kinetic law definition being used by a reaction.

## Examples

This example shows how to create a kinetic law definition and add it to the user-defined library.

- 1 Create a kinetic law definition.

```
abstkineticlawObj = sbioabstractkineticlaw('ex_myLaw1', '(k1*s)/(k2+k1+s)');
```

- 2 Assign the parameter and species variables in the expression.

```
set (abstkineticlawObj, 'SpeciesVariables', {'s'});
set (abstkineticlawObj, 'ParameterVariables', {'k1', 'k2'});
```

- 3 Add the new kinetic law definition to the user-defined library.

```
sbioaddtolibrary(abstkineticlawObj);
```

The function adds the kinetic law definition to the user-defined library. You can verify this using `sbiowhos`.

```
sbiowhos -kineticlaw -userdefined
```

```
SimBiology Abstract Kinetic Law Array
```

Index:	Library:	Name:	Expression:
1	UserDefined	mylaw1	$(k1*s)/(k2+k1+s)$

- 4 Use the new kinetic law definition when defining a reaction's kinetic law.

```
modelObj = sbiomodel('cell');  
reactionObj = addreaction(modelObj, 'A + B <-> B + C');  
kineticlawObj = addkineticlaw(reactionObj, 'ex_mylaw1');
```

---

**Note** Remember to specify the `SpeciesVariableNames` and the `ParameterVariableNames` in `kineticlawObj` to fully define the `ReactionRate` of the reaction.

---

## See Also

[addkineticlaw](#) | [sbioabstractkineticlaw](#) | [sbioremovefromlibrary](#) | [sbioroot](#) | [sbiounit](#) | [sbiounitprefix](#)

**Introduced in R2006a**

# sbioconsmoiety

Find conserved moieties in SimBiology model

## Syntax

```
[G, Sp] = sbioconsmoiety(modelObj)
[G, Sp] = sbioconsmoiety(modelObj, alg)
H = sbioconsmoiety(modelObj, alg, 'p')
H = sbioconsmoiety(modelObj, alg, 'p', FormatArg)
[SI, SD, L0, NR, ND] = sbioconsmoiety(modelObj, 'link')
```

## Arguments

<i>G</i>	An m-by-n matrix, where m is the number of conserved quantities found and n is the number of species in the model. Each row of <i>G</i> specifies a linear combination of species whose rate of change over time is zero.
<i>Sp</i>	Cell array of species names that labels the columns of <i>G</i> .  If the species are in multiple compartments, species names are qualified with the compartment name in the form <code>compartmentName.speciesName</code> . For example, <code>nucleus.DNA</code> , <code>cytoplasm.mRNA</code> .
<i>modelObj</i>	Model object to be evaluated for conserved moieties.
<i>alg</i>	Specify algorithm to use during evaluation of conserved moieties. Valid values are 'qr', 'rreduce', or 'semipos'.
<i>H</i>	Cell array of character vectors containing the conserved moieties.
<i>p</i>	Prints the output according to the format defined by <i>FormatArg</i> .
<i>FormatArg</i>	Specifies formatting for the output <i>H</i> . <i>FormatArg</i> must either be a character vector or string specifying a C-style format, or a positive integer specifying the maximum number of digits of precision used.

<i>SI</i>	Cell array containing the names of independent species in the model.
<i>SD</i>	Cell array containing the names of dependent species in the model.
<i>L0</i>	<p>Link matrix relating <i>SI</i> and <i>SD</i>. The link matrix <i>L0</i> satisfies <math>ND = L0 * NR</math>. For the 'link' functionality, species with their <code>BoundaryCondition</code> or <code>ConstantAmount</code> properties set to true are treated as having stoichiometry of zero in all reactions.</p> <p><i>L0</i> is a sparse matrix. To convert it to a full matrix, use the <code>full</code> function.</p>
<i>NR</i>	<p>Reduced stoichiometry matrices containing one row for each independent species. The concatenated matrix <math>[NR; ND]</math> is a row-permuted version of the full stoichiometry matrix of <i>modelObj</i>.</p> <p><i>NR</i> is a sparse matrix. To convert it to a full matrix, use the <code>full</code> function.</p>
<i>ND</i>	<p>Reduced stoichiometry matrices containing one row for each dependent species. The concatenated matrix <math>[NR; ND]</math> is a row-permuted version of the full stoichiometry matrix of <i>modelObj</i>.</p> <p><i>ND</i> is a sparse matrix. To convert it to a full matrix, use the <code>full</code> function.</p>

## Description

$[G, Sp] = \text{sbioconsmoiety}(\text{modelObj})$  calculates a complete set of linear conservation relations for the species in the SimBiology model object *modelObj*.

`sbioconsmoiety` computes conservation relations by analyzing the structure of the model object's stoichiometry matrix. Thus, `sbioconsmoiety` does not include species that are governed by algebraic or rate rules.

$[G, Sp] = \text{sbioconsmoiety}(\text{modelObj}, \text{alg})$  provides an algorithm specification. For *alg*, specify 'qr', 'rreduce', or 'semipos'.



- When you specify 'qr', `sbioconsmoiety` uses an algorithm based on QR factorization. From a numerical standpoint, this is the most efficient and reliable approach.
- When you specify 'rreduce', `sbioconsmoiety` uses an algorithm based on row reduction, which yields better numbers for smaller models. This is the default.
- When you specify 'semipos', `sbioconsmoiety` returns conservation relations in which all the coefficients are greater than or equal to 0, permitting a more transparent interpretation in terms of physical quantities.

For larger models, the QR-based method is recommended. For smaller models, row reduction or the semipositive algorithm may be preferable. For row reduction and QR factorization, the number of conservation relations returned equals the row rank degeneracy of the model object's stoichiometry matrix. The semipositive algorithm may return a different number of relations. Mathematically speaking, this algorithm returns a generating set of vectors for the space of semipositive conservation relations.

`H = sbioconsmoiety(modelObj, alg, 'p')` returns a cell array of character vectors `H` containing the conserved quantities in `modelObj`.

`H = sbioconsmoiety(modelObj, alg, 'p', FormatArg)` specifies formatting for the output `H`. `FormatArg` should either be a C-style format string, or a positive integer specifying the maximum number of digits of precision used.

`[SI, SD, L0, NR, ND] = sbioconsmoiety(modelObj, 'link')` uses a QR-based algorithm to compute information relevant to the dimensional reduction, via conservation relations, of the reaction network in `modelObj`.

## Examples

### Example 1

This example shows conserved moieties in a cycle.

- 1 Create a model with a cycle. For convenience use arbitrary reaction rates, as this will not affect the result.

```
modelObj = sbiomodel('cycle');
modelObj.addreaction('a -> b', 'ReactionRate', '1');
modelObj.addreaction('b -> c', 'ReactionRate', 'b');
modelObj.addreaction('c -> a', 'ReactionRate', '2*c');
```

2 Look for conserved moieties.

```
[g sp] = sbioconsmoiety(modelObj)

g =

     1     1     1

sp =

     'a'
     'b'
     'c'
```

## Example 2

Explore semipositive conservation relations in the `oscillator` model.

```
modelObj = sbmlimport('oscillator');
sbioconsmoiety(modelObj, 'semipos', 'p')

ans =

     'pol + pol_0pA + pol_0pB + pol_0pC'
     '0pB + pol_0pB + pA_0pB1 + pA_0pB_pA + pA_0pB2'
     '0pA + pol_0pA + pC_0pA1 + pC_0pA2 + pC_0pA_pC'
     '0pC + pol_0pC + pB_0pC1 + pB_0pC2 + pB_0pC_pB'
```

## See Also

`getstoichmatrix`

## Topics

“Conserved Moiety Determination”

**Introduced in R2006a**

# sbioconvertunits

Convert unit and unit value to new unit

## Syntax

```
sbioconvertunits(Obj, 'unit')
```

## Description

`sbioconvertunits(Obj, 'unit')` converts the current `*Units` property on SimBiology object, *Obj* to the unit, *unit*. This function configures the `*Units` property to *unit* and updates the corresponding value property. For example, `sbioconvertunits` on a `speciesObj` updates the `InitialAmount` property value and the `InitialAmountUnits` property value.

*Obj* can be an array of SimBiology objects. *Obj* must be a SimBiology object that contains a unit property. The SimBiology objects that contain a unit property are compartment, parameter, and species objects. For example, if *Obj* is a species object with `InitialAmount` configured to 1 and `InitialAmountUnits` configured to mole, after the call to `sbioconvertunits` with *unit* specified as molecule, `speciesObj` `InitialAmount` is 6.0221e23 and `InitialAmountUnits` is molecule.

## Examples

Convert the units of the initial amount of glucose from molecule to mole.

- 1 Create the species 'glucose' and assign an initial amount of 23 molecule.

At the command prompt, type:

```
modelObj = sbiomodel('cell');
compObj = addcompartment(modelObj, 'C');
speciesObj = addspecies(compObj, 'glucose', 23, 'InitialAmountUnits', 'molecule')
```

SimBiology Species Array

```
Index:      Compartment:  Name:      InitialAmount:  InitialAmountUnits:
1          C             glucose    23             molecule
```

- 2 Convert the `InitialAmountUnits` of glucose from molecule to mole.

```
sbioconvertunits (speciesObj, 'mole')
```

- 3 Verify the conversion of units and `InitialAmount` value.

Units are converted from molecule to mole.

```
get (speciesObj, 'InitialAmountUnits')
```

```
ans =
```

```
mole
```

The `InitialAmount` value is changed.

```
get (speciesObj, 'InitialAmount')
```

```
ans =
```

```
3.8192e-023
```

## See Also

`sbioshowunits`

## Topics

`sbioshowunits`

**Introduced in R2006a**

# sbiocopylibrary

Copy library to disk

## Syntax

```
sbiocopylibrary ('kineticlaw', 'LibraryFileName')  
sbiocopylibrary ('unit', 'LibraryFileName')
```

## Description

`sbiocopylibrary ('kineticlaw', 'LibraryFileName')` copies all user-defined kinetic law definitions to the file `LibraryFileName.sbklib` and places the copied file in the current directory.

`sbiocopylibrary ('unit', 'LibraryFileName')` copies all user-defined units and unit prefixes to the file `LibraryFileName.sbulib`.

To get the kinetic law definitions that are in the built-in or user-defined libraries, first create a root object using `sbiroot`, then use the commands `get(rootObj.BuiltInLibrary, 'KineticLaws')` or `get(rootObj.UserDefinedLibrary, 'KineticLaws')`.

To add a kinetic law definition to the user-defined library, use `sbioaddtolibrary`.

To add a unit to the user-defined library, use `sbiounit` followed by `sbioaddtolibrary`. To add a unit prefix to the user-defined library, use `sbiounitprefix` followed by `sbioaddtolibrary`.

## Examples

Create a kinetic law definition, add it to the user-defined library, and then copy the user-defined kinetic law library to a `.sbklib` file.

- 1 Create a kinetic law definition.

```
abstkineticlawObj = sbioabstractkineticlaw('mylaw1', '(k1*s)/(k2+k1+s)');
```

- 2 Add the new a kinetic law definition to the user-defined library.

```
sbioaddtolibrary(abstkineticlawObj);
```

`sbioaddtolibrary` adds the kinetic law definition to the user-defined library. You can verify this using `sbiowhos`.

```
sbiowhos -kineticlaw -userdefined
```

```
SimBiology Abstract Kinetic Law Array
```

Index:	Library:	Name:	Expression:
1	UserDefined	mylaw1	(k1*s)/(k2+k1+s)

- 3 Copy the user-defined kinetic law library.

```
sbiocopylibrary ('kineticlaw', 'myLibFile')
```

- 4 Verify with `sbiowhos`.

```
sbiowhos -kineticlaw myLibFile
```

## See Also

`sbioabstractkineticlaw` | `sbioaddtolibrary` | `sbioremovefromlibrary` | `sbiounit` | `sbiounitprefix`

**Introduced in R2006a**

# sbiodesktop

Open SimBiology desktop for modeling and simulation

## Syntax

```
sbiodesktop  
sbiodesktop(modelObj)  
sbiodesktop(File)
```

## Input Arguments

<i>modelObj</i>	SimBiology model object or an array of model objects.
<i>File</i>	String specifying a file name or path and file name of an sbproj file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.

## Description

sbiodesktop opens the SimBiology desktop, which lets you:

- Build a SimBiology model by representing reaction pathways and entering kinetic data for the reactions.
- Import or export SimBiology models to and from the MATLAB workspace or from a Systems Biology Markup Language (SBML) file.
- Modify an existing SimBiology model.
- Simulate a SimBiology model through individual or ensemble runs.
- View results from the simulation.
- Perform analysis tasks such as sensitivity analysis, parameter and species scans, and calculation of conserved moieties.
- Create and/or modify user-defined units and unit prefixes.

- Create and/or modify user-defined kinetic laws.

`sbiodesktop(modelObj)` opens the SimBiology desktop with *modelObj*, a SimBiology model object. If there is a project open in the SimBiology desktop, this command adds *modelObj* to the project.

`sbiodesktop(File)` opens the project specified by *File* in the SimBiology desktop. *File* is a string specifying a file name or path and file name of an sbproj file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. If a project is open in the desktop, the software replaces it with the new project, after prompting you to save any changes.

The `Parent` property of a SimBiology model object is set to the SimBiology root object. The root object contains a list of model objects that are accessible from the MATLAB command line and from the SimBiology desktop. Because both the command line and the desktop point to the same model object in the `Root` object, changes you make to the model at the command line are reflected in the desktop, and vice versa.

---

**Note** The `sbioreset` command removes all models from the root object. Therefore, this command also removes all models from the SimBiology desktop.

---

## Examples

Create a SimBiology model in the MATLAB workspace, and then open the SimBiology desktop with the model:

```
modelObj = sbiomodel('cell');  
sbiodesktop(modelObj)
```

## See Also

`sbiofittool` | `sbioroot` | `simbiology`

**Introduced before R2006a**



## sbiodose

Construct dose object

### Syntax

```
dose = sbiodose(DoseName)
dose = sbiodose(DoseName,DoseType)
dose = sbiodose(DoseName,Name,Value)
```

### Description

`dose = sbiodose(DoseName)` creates a `RepeatDose` object and sets its `Name` property to `DoseName`.

`dose = sbiodose(DoseName,DoseType)` creates either a `RepeatDose` object or `ScheduleDose` object based on `DoseType`.

`dose = sbiodose(DoseName,Name,Value)` uses name-value pair arguments to define the properties of the dose object. You can enter the name-value pairs in the same format supported by the function set. Use the `get` function to view all the properties of the object.

### Examples

#### Increase Drug Concentration in a One-Compartment Model via First-order Dosing

This example shows how to set up a dosing regimen that follows the first-order absorption kinetics.

##### Background

Suppose you have a one-compartment model with a species named `drug` that represents the total amount of drug in the body. The drug is added to the body via the first-order dosing represented by the reaction `dose -> drug`, with the absorption rate constant `ka`.

It is removed from the body via the first-order elimination represented by the reaction `drug -> null`, with the elimination rate constant `ke`. This example shows how to set up such a one-compartment model, the first-order absorption and elimination.

## Create a One-compartment Model

Create a SimBiology model named `onecomp`.

```
m1 = sbiomodel('onecomp');
```

Define the drug elimination by adding a reaction `drug -> null` to the model. The `drug` species represents the total amount of drug in the compartment.

```
r1 = addreaction(m1, 'drug -> null');
```

Note that a compartment and the species `drug` are automatically created, and `drug` is added to the compartment. The `null` species is a reserved species that acts as a sink in this reaction.

Add a mass action kinetic law to the reaction. This kinetic law defines the drug elimination to follow the first-order kinetics.

```
k1 = addkineticlaw(r1, 'MassAction');
```

Define the elimination rate parameter `ke` and add it to the kinetic law.

```
p1 = addparameter(k1, 'ke', 'Value', 1.0, 'ValueUnits', '1/hour');
```

Specify the rate parameter `ke` as the forward rate parameter of the reaction by setting the `ParameterVariableNames` property of kinetic law object `k2`. This allows SimBiology to determine the reaction rate for `drug -> null` reaction.

```
k1.ParameterVariableNames = 'ke';
```

## Set up the First-order Dosing

Add a reaction that represents the drug absorption using the second species `dose`. It represents an intermediate species that will be dosed directly and is required to set up the first-order absorption kinetics.

```
r2 = addreaction(m1, 'dose -> drug');
```

Add a mass action kinetic law to the reaction. This kinetic law defines the drug absorption to follow the first-order kinetics.

```
k2 = addkineticlaw(r2, 'MassAction');
```

Define the absorption rate parameter ka and add it to the kinetic law.

```
p2 = addparameter(k2, 'ka', 'Value', 0.1, 'ValueUnits', '1/hour');
```

Specify the rate parameter ka as the forward rate parameter of the reaction by setting the `ParameterVariableNames` property of kinetic law object k1. This allows SimBiology to determine the reaction rate for dose -> drug reaction.

```
k2.ParameterVariableNames = 'ka';
```

Suppose you want to increase the drug concentration in the system by administering a series of doses: 250 mg three times a day (t.i.d) for two days. Specify the amount of the dose (`Amount`), the time interval between each dose (`Interval`), and the total number of doses (`RepeatCount`). You also need to set the `Active` property of the dose object to `true` so that the dose will be applied to the model during simulation. `RepeatCount` was set to 5, instead of 6 since it represents the number of doses *after* the first dose at the default dose start time (`d1.StartTime = 0`).

```
d1 = sbiodose('d1', 'repeat');
d1.Amount = 250;
d1.AmountUnits = 'milligram';
d1.Interval = 8;
d1.TimeUnits = 'hour';
d1.RepeatCount = 5;
d1.Active = true;
```

Specify the target species of the dose object. The target must be the dose species, not the drug species, so that the drug absorption follows the first-order kinetics.

```
d1.TargetName = 'dose';
```

### Simulate the Model

Change the simulation stop time to 48 hours to match the dosing schedule.

```
cs = getconfigset(m1);
cs.StopTime = 48;
cs.TimeUnits = 'hour';
```

In addition, do not log the dose species data as you are mainly interested in monitoring the drug species which is the drug concentration in the system. This makes visualizing the species in a plot more convenient. To accomplish this, set the `StatesToLog` property to include the species drug only.

```
cs.RuntimeOptions.StatesToLog = {'drug'};
```

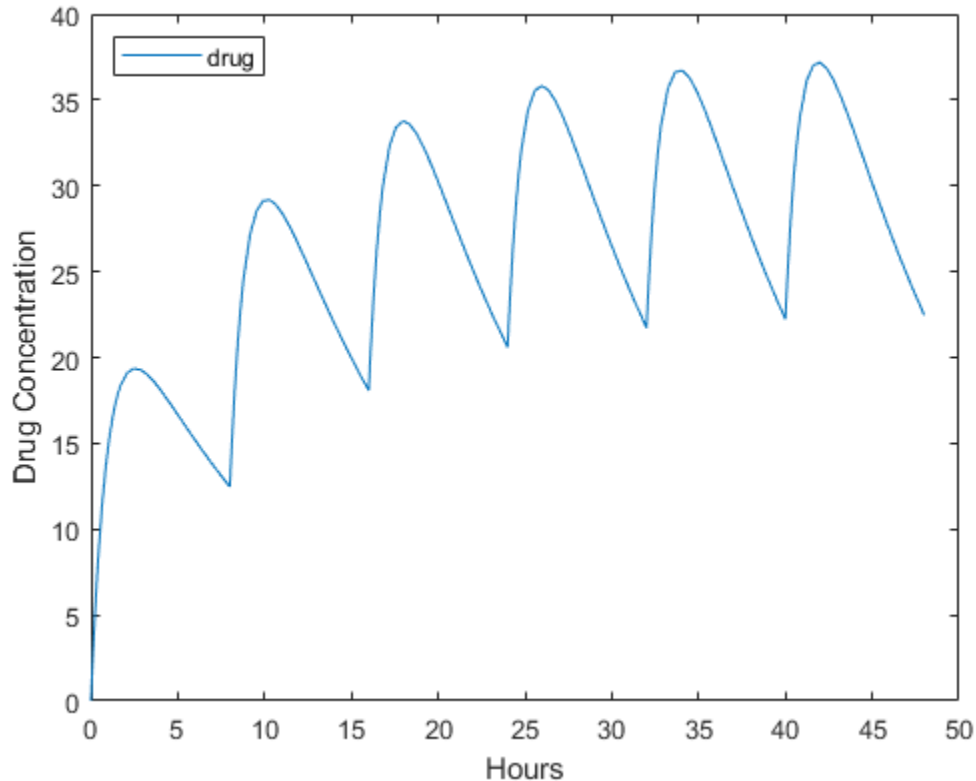
Simulate the model using the dosing schedule defined by the |d1 |dose object.

```
[t,sd,species] = sbiosimulate(m1,d1);
```

### **Plot Results**

Plot the concentration versus the time profile of the drug in the compartment.

```
plot(t,sd);  
legend(species,'Location','NorthWest');  
xlabel('Hours');  
ylabel('Drug Concentration');
```



### Add a Series of Bolus Doses to a One-Compartment Model

This example shows how to add a series of bolus doses to one-compartment model.

#### Background

Suppose you have a one-compartment model with a species named drug that represents the total amount of drug in the body. The drug is removed from the body via the first-order elimination represented by the reaction  $\text{drug} \rightarrow \text{null}$ , with the elimination rate constant  $k_e$ . In other words, the drug concentration versus the time profile follows the

monoexponential decline  $C_t = C_0 e^{-k_e t}$ , where  $C_t$  is the drug concentration at time  $t$ ,  $C_0$  is

the initial concentration, and  $k_e$  is the elimination rate constant. This example shows how to set up such a one-compartment model and administer a series of bolus doses, namely 250 mg three times a day (tid) for two days.

### Create a One-compartment Model

First create a SimBiology model named `onecomp`.

```
m1 = sbiomodel('onecomp');
```

Define the elimination of the drug from the system by adding a reaction `drug -> null` to the model.

```
r1 = addreaction(m1, 'drug -> null');
```

The species `drug` is automatically created and the reaction is added to the compartment. The `null` species is a reserved species that acts as a sink in this reaction.

Add a mass action kinetic law to the reaction. This kinetic law defines the drug elimination to follow the first-order kinetics.

```
k1 = addkineticlaw(r1, 'MassAction');
```

Define the elimination rate parameter `ke` and add it to the kinetic law.

```
p1 = addparameter(k1, 'ke', 'Value', 1.0, 'ValueUnits', '1/hour');
```

Specify the rate parameter `ke` as the forward rate parameter of the reaction by setting the `ParameterVariableNames` property of kinetic law object `k1`. This allows SimBiology to determine the reaction rate for `drug -> null` reaction.

```
k1.ParameterVariableNames = 'ke';
```

### Set up a Series of Bolus Doses

Suppose you want to increase the drug concentration in the system by administering a series of bolus doses: 250 mg three times a day (tid) for two days. Create a repeat dose object. Specify the amount of the dose (`Amount`), the dose target, the time interval between each dose (`Interval`), and the total number of doses (`RepeatCount`). You also need to set the `Active` property of the dose object to `true` so that the dose is applied to the model during simulation.

```
d1 = sbiodose('d1', 'repeat');  
d1.Amount = 250;
```

```
d1.AmountUnits = 'milligram';  
d1.TargetName = 'drug';  
d1.Interval = 8;  
d1.TimeUnits = 'hour';  
d1.RepeatCount = 5;  
d1.Active = true;
```

RepeatCount was set to 5, instead of 6 since it represents the number of doses *after* the first dose at the default dose start time (`d1.StartTime = 0`).

### Simulate the Model

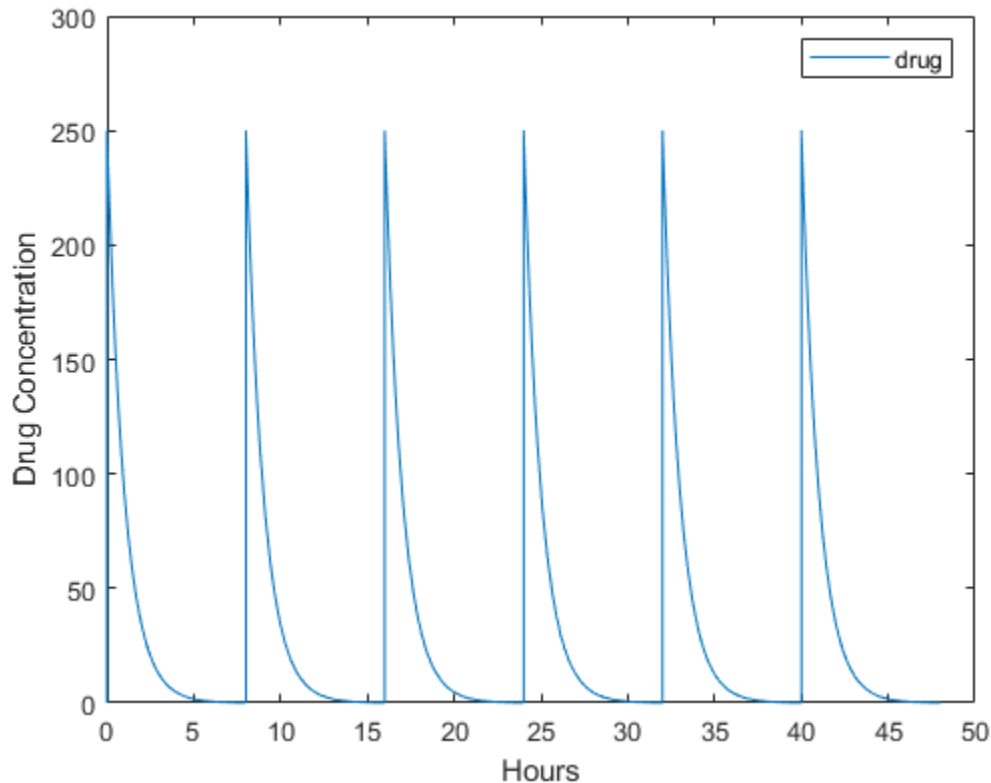
Change the simulation stop time to 48 hours to match the dosing schedule defined by the `d1` dose object.

```
cs = getconfigset(m1);  
cs.StopTime = 48;  
cs.TimeUnits = 'hour';  
[t,sd,species] = sbiosimulate(m1,d1);
```

### Plot Results

Plot the concentration versus the time profile of the drug in the system.

```
plot(t,sd);  
legend(species);  
xlabel('Hours');  
ylabel('Drug Concentration');
```



### **Increase Drug Concentration in a One-Compartment Model via Zero-order Dosing**

This example shows how to set up a dosing regimen that follows the zero-order absorption kinetics.

#### **Background**

Suppose you have a one-compartment model with a species named `drug` that represents the total amount of drug in the body. The drug is removed from the body via the first-order elimination represented by the reaction `drug -> null`, with the elimination rate constant  $k_e$ . In other words, the drug concentration versus the time profile follows the



monoexponential decline  $C_t = C_0 e^{-k_e t}$ , where  $C_t$  is the drug concentration at time  $t$ ,  $C_0$  is the initial concentration, and  $k_e$  is the elimination rate constant. This example shows how to set up such a one-compartment model and increase the drug concentration in the compartment via the zero-order absorption that takes 25 hours to administer the total dose amount of 250 mg.

### Create a One-compartment Model

Create a SimBiology model named `onecomp`.

```
m1 = sbiomodel('onecomp');
```

Define the elimination of the drug from the system by adding a reaction `drug -> null` to the model.

```
r1 = addreaction(m1,'drug -> null');
```

The species `drug` is automatically created and added to the compartment. The `null` species is a reserved species that acts as a sink in this reaction.

Add a mass action kinetic law to the reaction. This kinetic law defines the drug elimination to follow the first-order kinetics.

```
k1 = addkineticlaw(r1,'MassAction');
```

Define the elimination rate parameter `ke` and add it to the kinetic law.

```
p1 = addparameter(k1,'ke','Value',1.0,'ValueUnits','1/hour');
```

Specify the rate parameter `ke` as the forward rate parameter of the reaction by setting the `ParameterVariableNames` property of kinetic law object `k1`. This allows SimBiology to determine the reaction rate for `drug -> null` reaction.

```
k1.ParameterVariableNames = 'ke';
```

### Set up Zero-order Dosing

To set up zero-order dosing, first create a zero-order duration parameter `p2` that represents the time it takes to administer a dose. Next, specify the amount of the dose (`Amount`), the dose target (`TargetName`), and the name of the zero-order duration parameter (`DurationParameterName`). You also need to set the `Active` property of the dose object to `true` so that the dose is applied to the model during simulation.

```
p2 = addparameter(m1, 'duration', 'Value', 25, 'ValueUnits', 'hour');
d1 = sbiodose('d1');
d1.Amount = 250;
d1.AmountUnits = 'milligram';
d1.TargetName = 'drug';
d1.DurationParameterName = 'duration'; %Name of the duration parameter |p2|
d1.Active = true;
```

## Simulate the Model

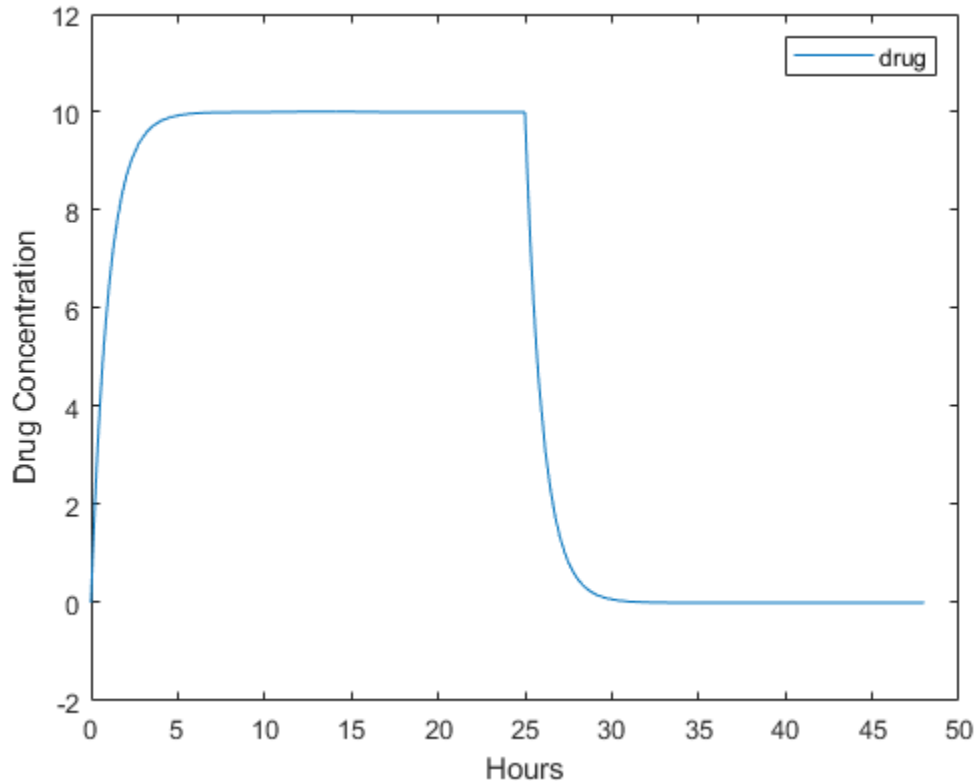
Change the simulation stop time to 48 hours to see the complete time profile. Apply the dosing schedule defined by d1 to the model during simulation.

```
cs = getconfigset(m1);
cs.StopTime = 48;
cs.TimeUnits = 'hour';
[t, sd, species] = sbiosimulate(m1, d1);
```

## Plot results

Plot the concentration versus the time profile of the drug in the compartment.

```
plot(t, sd);
legend(species);
xlabel('Hours');
ylabel('Drug Concentration');
```



### Add an Infusion Dose to a One-Compartment Model

This example shows how to add a constant-rate infusion dose to one-compartment model.

#### Background

Suppose you have a one-compartment model with a species named drug that represents the total amount of drug in the body. The drug is removed from the body via the first-order elimination represented by the reaction  $\text{drug} \rightarrow \text{null}$ , with the elimination rate constant  $k_e$ . In other words, the drug concentration versus the time profile follows the

monoexponential decline  $C_t = C_0 e^{-k_e t}$ , where  $C_t$  is the drug concentration at time  $t$ ,  $C_0$  is

the initial concentration, and  $k_e$  is the elimination rate constant. This example shows how to set up such a one-compartment model and add an infusion dose at a constant rate of 10 mg/hour for the total dose amount of 250 mg.

## Create a One-compartment Model

Create a SimBiology model named `onecomp`.

```
m1 = sbiomodel('onecomp');
```

Define the elimination of the drug from the system by adding a reaction `drug -> null` to the model.

```
r1 = addreaction(m1, 'drug -> null');
```

The species `drug` is automatically created and added to the compartment. The `null` species is a reserved species that acts as a sink in this reaction.

Add a mass action kinetic law to the reaction. This kinetic law defines the drug elimination to follow the first-order kinetics.

```
k1 = addkineticlaw(r1, 'MassAction');
```

Define the elimination rate parameter  $k_e$  and add it to the kinetic law.

```
p1 = addparameter(k1, 'ke', 'Value', 1.0, 'ValueUnits', '1/hour');
```

Specify the rate parameter  $k_e$  as the forward rate parameter of the reaction by setting the `ParameterVariableNames` property of kinetic law object `k1`. This allows SimBiology to determine the reaction rate for `drug -> null` reaction.

```
k1.ParameterVariableNames = 'ke';
```

## Set up an Infusion Dose

Specify the amount of the dose (`Amount`), the dose target (`TargetName`), and the infusion rate (`Rate`). You also need to set the `Active` property of the dose object to `true` so that the dose is applied to the model during simulation.

```
d1 = sbiodose('d1');  
d1.Amount = 250;  
d1.TargetName = 'drug';  
d1.Rate = 10;  
d1.RateUnits = 'milligram/hour';  
d1.Active = true;
```

## Simulate the Model

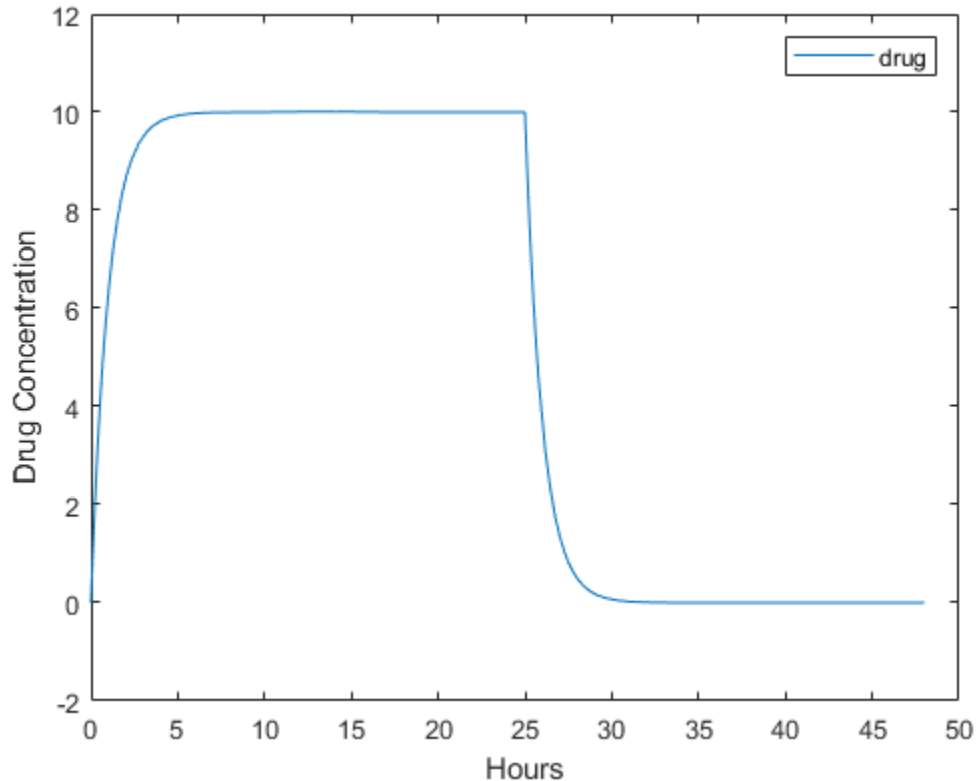
Change the simulation stop time to 48 hours to see the complete time course. Apply the dosing schedule defined by d1 to the model during simulation.

```
cs = getconfigset(m1);  
cs.StopTime = 48;  
cs.TimeUnits = 'hour';  
[t,sd,species] = sbiosimulate(m1,d1);
```

## Plot results

Plot the concentration versus the time profile of the drug in the system.

```
plot(t,sd);  
legend(species);  
xlabel('Hours');  
ylabel('Drug Concentration');
```



## Input Arguments

### **DoseName** — Name of the dose object

character vector | string

Name of the dose object, specified as a character vector or string.

Example: '250mg\_tid'

Data Types: char

### **DoseType** — Type of the dose object

'schedule' | 'repeat'

Type of the dose object, specified as 'schedule' for a ScheduleDose object and 'repeat' for a RepeatDose object.

Example: 'schedule'

Data Types: char

## Output Arguments

### **dose** — Dose object

RepeatDose object | ScheduleDose object

Dose object, returned as a RepeatDose object or ScheduleDose object.

## See Also

RepeatDose object | ScheduleDose object | adddose | copyobj | get | getdose | removedose | set

## Topics

Model  
"Doses"

**Introduced in R2010a**

## sbioensembleplot

Show results of ensemble run using 2-D or 3-D plots

### Syntax

```
sbioensembleplot(simdataObj)
sbioensembleplot(simdataObj, Names)
sbioensembleplot(simdataObj, Names, Time)
```

```
FH = sbioensembleplot(simdataObj, Names)
FH = sbioensembleplot(simdataObj, Names, Time)
```

### Arguments

<i>simdataObj</i>	An object that contains simulation data. You can generate a <i>simdataObj</i> object using the function <code>sbioenssemblerun</code> . All elements of <i>simdataObj</i> must contain data for the same states in the same model.
<i>Names</i>	Character vector, string, string vector, string array, or cell array of character vectors. <i>Names</i> may include qualified names such as ' <i>CompartmentName.SpeciesName</i> ' or ' <i>ReactionName.ParameterName</i> ' to resolve ambiguities. Specifying {} or empty string array ( <code>string.empty</code> ) for <i>Names</i> plots data for all states contained in <i>simdataObj</i> .
<i>Time</i>	A numeric scalar value. If the specified <i>Time</i> is not an element of the time vectors in <i>simdataObj</i> , then the function resamples <i>simdataObj</i> as necessary using linear interpolation.
<i>FH</i>	Array of handles to figure windows.

### Description

`sbioensembleplot(simdataObj)` shows a 3-D shaded plot of time-varying distribution of all logged states in the SimData array *simdataObj*. The `sbioenssemblerun` function



plots an approximate distribution created by fitting a normal distribution to the data at every time step.

`sbioensembleplot(simdataObj, Names)` plots the distribution for the data specified by *Names*.

`sbioensembleplot(simdataObj, Names, Time)` plots a 2-D histogram of the actual data of the ensemble distribution of the states specified by *Names* at the particular time point *Time*.

`FH = sbioensembleplot(simdataObj, Names)` returns an array of handles *FH*, to the figure window for the 3-D distribution plot.

`FH = sbioensembleplot(simdataObj, Names, Time)` returns an array of handles *FH*, to the figure window for the 2-D histograms.

## Examples

This example shows how to plot data from an ensemble run without interpolation.

- 1 The project file, `radiodecay.sbproj`, contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.

```
sbioloadproject('radiodecay.sbproj', 'm1');
```

- 2 Change the solver of the active configuration set to be `ssa`. Also, adjust the `LogDecimation` property on the `SolverOptions` property of the configuration set to reduce the size of the data generated.

```
cs = getconfigset(m1, 'active');
set(cs, 'SolverType', 'ssa');
so = get(cs, 'SolverOptions');
set(so, 'LogDecimation', 10);
```

- 3 Perform an ensemble of 20 runs with no interpolation.

```
simdataObj = sbioensemblerrun(m1, 20);
```

- 4 Create a 2-D distribution plot of the species 'z' at time = 1.0.

```
FH1 = sbioensembleplot(simdataObj, 'z', 1.0);
```

- 5 Create a 3-D shaded plot of both species.

```
FH2 = sbioensembleplot(simdataObj, {'x', 'z'});
```

## **See Also**

[sbioensemblerrun](#) | [sbioensemblestats](#) | [sbiomodel](#)

**Introduced in R2006a**

# sbioensemblerun

Multiple stochastic ensemble runs of SimBiology model

## Syntax

```

simdataObj = sbioensemblerun(modelObj, Numruns)
simdataObj = sbioensemblerun(modelObj, Numruns, Interpolation)
simdataObj = sbioensemblerun(modelObj, Numruns, configsetObj)
simdataObj = sbioensemblerun(modelObj, Numruns, configsetObj,
Interpolation)
simdataObj = sbioensemblerun(modelObj, Numruns, variantObj)
simdataObj = sbioensemblerun(modelObj, Numruns, variantObj,
Interpolation)
simdataObj = sbioensemblerun(modelObj, Numruns, configsetObj,
variantObj)
simdataObj = sbioensemblerun(modelObj, Numruns, configsetObj,
variantObj, Interpolation)

```

## Arguments

<i>simdataObj</i>	An array of SimData objects containing simulation data generated by sbioensemblerun. All elements of <i>simdataObj</i> contain data for the same states in the same model.
<i>modelObj</i>	Model object to be simulated.
<i>Numruns</i>	Integer scalar representing the number of stochastic runs to make.
<i>Interpolation</i>	Character vector or string denoting the interpolation scheme to be used if data should be interpolated to get a consistent time vector. Valid values are 'linear' (linear interpolation), 'zoh' (zero-order hold), or 'off' (no interpolation). Default is 'off'. If interpolation is on, the data is interpolated to match the time vector with the smallest simulation stop time.

<i>configsetObj</i>	Specify the configuration set object to use in the ensemble simulation. For more information about configuration sets, see <code>Configset</code> object.
<i>variantObj</i>	Specify the variant object to apply to the model during the ensemble simulation. For more information about variant objects, see <code>Variant</code> object.

## Description

*simdataObj* = `sbioensemblerrun(modelObj, Numruns)` performs a stochastic ensemble run of the SimBiology model object (*modelObj*), and returns the results in *simdataObj*, an array of `SimData` objects. The active `configset` and the active variants are used during simulation and are saved in the output, `SimData` object (*simdataObj*).

`sbioensemblerrun` uses the settings in the active `configset` on the model object (*modelObj*) to perform the repeated simulation runs. The `SolverType` property of the active `configset` must be set to one of the stochastic solvers: 'ssa', 'expltau', or 'impltau'. `sbioensemblerrun` generates an error if the `SolverType` property is set to any of the deterministic (ODE) solvers.

*simdataObj* = `sbioensemblerrun(modelObj, Numruns, Interpolation)` performs a stochastic ensemble run of a model object (*modelObj*), and interpolates the results of the ensemble run onto a common time vector using the interpolation scheme (*Interpolation*).

*simdataObj* = `sbioensemblerrun(modelObj, Numruns, configsetObj)` performs an ensemble run of a model object (*modelObj*), using the specified configuration set (*configsetObj*).

*simdataObj* = `sbioensemblerrun(modelObj, Numruns, configsetObj, Interpolation)` performs an ensemble run of a model object (*modelObj*), using the specified configuration set (*configsetObj*), and interpolates the results of the ensemble run onto a common time vector using the interpolation scheme (*Interpolation*).

*simdataObj* = `sbioensemblerrun(modelObj, Numruns, variantObj)` performs an ensemble run of a model object (*modelObj*), using the variant object or array of variant objects (*variantObj*).

*simdataObj* = `sbioensemblerrun(modelObj, Numruns, variantObj, Interpolation)` performs an ensemble run of a model object (*modelObj*), using the

variant object or array of variant objects (*variantObj*), and interpolates the results of the ensemble run onto a common time vector using the interpolation scheme (*Interpolation*).

`simdataObj = sbioensemblerun(modelObj, Numruns, configsetObj, variantObj)` performs an ensemble run of a model object (*modelObj*), using the configuration set (*configsetObj*), and the variant object or array of variant objects (*variantObj*). If the configuration set object (*configsetObj*) is empty, the active configset on the model is used for simulation. If the variant object (*variantObj*) is empty, then no variant (not even the active variants in the model) is used for the simulation.

`simdataObj = sbioensemblerun(modelObj, Numruns, configsetObj, variantObj, Interpolation)` performs an ensemble run of a model object (*modelObj*), using the configuration set (*configsetObj*), and the variant object or array of variant objects (*variantObj*), and interpolates the results of the ensemble run onto a common time vector using the interpolation scheme (*Interpolation*).

## Examples

This example shows how to perform an ensemble run and generate a 2-D distribution plot.

- 1 The project file, `radiodecay.sbproj`, contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.  

```
sbioloadproject('radiodecay.sbproj','m1');
```
- 2 Change the solver of the active configset to be `ssa`. Also, adjust the `LogDecimation` property on the `SolverOptions` property of the configuration set.

```
cs = getconfigset(m1, 'active');  
set(cs, 'SolverType', 'ssa');  
so = get(cs, 'SolverOptions');  
set(so, 'LogDecimation', 10);
```

---

**Tip** The `LogDecimation` property lets you define how often the simulation data is recorded as output. If your model has high concentrations or amounts of species, or a long simulation time (for example, 600s), you can record simulation data less often to manage the amount of data generated. Be aware that by doing so you might miss some transitions if your model is very dynamic. Try setting `LogDecimation` to 10 or more.

---

- 3 Perform an ensemble of 20 runs with linear interpolation to get a consistent time vector.

```
simdata = sbioensamplerun(m1, 20, 'linear');
```

- 4 Create a 2-D distribution plot of the species 'z' at a time = 1.0.

```
FH = sbioensembleplot(simdata, 'z', 1.0);
```

### See Also

[SimData object](#) | [addconfigset](#) | [getconfigset](#) | [sbioensembleplot](#) | [sbioensemblestats](#) | [setactiveconfigset](#)

**Introduced in R2006a**

# sbioensemblestats

Get statistics from ensemble run data

## Syntax

```
[t,m] = sbioensemblestats(simDataObj)
[t,m,v] = sbioensemblestats(simDataObj)
[t,m,v,n] = sbioensemblestats(simDataObj)
[t,m,v,n] = sbioensemblestats(simDataObj,names)
[t,m,v,n] = sbioensemblestats(simDataObj,names,interpolation)
```

## Arguments

<i>t</i>	Column vector of time points
<i>m</i>	Matrix of mean values from the ensemble data. The number of rows in <i>m</i> is the length of the time vector <i>t</i> and the number of columns is equal to the number of species.
<i>simDataObj</i>	A cell array of SimData objects, where each SimData object holds data for a separate simulation run. All elements of <i>simDataObj</i> must contain data for the same states in the same model. When the time vectors of the elements of <i>simDataObj</i> are not identical, <i>simDataObj</i> is first resampled onto a common time vector (see <i>interpolation</i> below).
<i>v</i>	Matrix of variance obtained from the ensemble data. <i>v</i> has the same dimensions as <i>m</i> .
<i>n</i>	Cell array of character vectors for the quantity names whose mean and variance are returned in <i>m</i> and <i>v</i> , respectively. The number of elements in <i>n</i> is the same as the number of columns of <i>m</i> and <i>v</i> . The order of names in <i>n</i> corresponds to the order of columns of <i>m</i> and <i>v</i> .

<i>names</i>	Character vector, string, string vector, string array, or cell array of character vectors. <i>names</i> may include qualified names such as ' <i>CompartmentName.SpeciesName</i> ' or ' <i>ReactionName.ParameterName</i> ' to resolve ambiguities. If you specify empty {} or empty string array ( <code>string.empty</code> ) for <i>names</i> , <code>sbioensemblestats</code> returns statistics on all time courses contained in <i>simDataObj</i> .
<i>interpolation</i>	Character vector or string denoting the interpolation method to use for resampling of the data onto a common time vector with the smallest simulation stop time. See <code>resample</code> for a list of interpolation methods. Default is 'linear'.

## Description

$[t, m] = \text{sbioensemblestats}(simDataObj)$  computes the time-dependent ensemble mean  $m$  of the ensemble data *simDataObj*. If the time vectors of the ensemble data are not identical, by default, the function uses the 'linear' interpolation method to resample the data onto the common time vector. See `resample` for a list of interpolation methods.

$[t, m, v] = \text{sbioensemblestats}(simDataObj)$  also returns the variance  $v$  for the ensemble run data *simDataObj*.

$[t, m, v, n] = \text{sbioensemblestats}(simDataObj)$  also returns the names of quantities  $n$  corresponding to the mean  $m$  and variance  $v$  columns. Each column of  $m$  or  $v$  describes the ensemble mean or variance of a quantity (or state) as a function of time.

$[t, m, v, n] = \text{sbioensemblestats}(simDataObj, names)$  computes statistics only for the quantities specified by *names*.

$[t, m, v, n] = \text{sbioensemblestats}(simDataObj, names, interpolation)$  uses the interpolation method *interpolation* to resample the simulation data to have a consistent time vector. If the time vectors of the ensemble data are not identical and if you do not specify any interpolation method, the function uses the 'linear' interpolation method by default.



## Examples

The project file, `radiodecay.sbproj`, contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.

- 1 Load a SimBiology model `m1` from a SimBiology project file.

```
sbioloadproject('radiodecay.sbproj','m1');
```

- 2 Change the solver of the active configuration set to be `ssa`. Also, adjust the `LogDecimation` property on the `SolverOptions` property of the configuration set.

```
cs = getconfigset(m1, 'active');  
set(cs, 'SolverType', 'ssa');  
so = get(cs, 'SolverOptions');  
set(so, 'LogDecimation', 10);
```

- 3 Perform an ensemble of 20 runs with no interpolation.

```
simDataObj = sbioensemblerrun(m1, 20);
```

- 4 Get ensemble statistics for all species using the default interpolation method.

```
[T,M,V] = sbioensemblestats(simDataObj);
```

- 5 Get ensemble statistics for a specific species using the default interpolation scheme.

```
[T2,M2,V2] = sbioensemblestats(simDataObj, {'z'});
```

## See Also

`sbioensembleplot` | `sbioensemblerrun` | `sbiomodel`

**Introduced in R2006a**

## sbiofit

Perform nonlinear least-squares regression

---

**Note** Statistics and Machine Learning Toolbox™, Optimization Toolbox™, and Global Optimization Toolbox are recommended for this function.

---

### Syntax

```
fitResults = sbiofit(sm,grpData,responseMap,estiminfo)
fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing)
fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing,
functionName)
fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing,
functionName,options)
fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing,
functionName,options,variants)
fitResults = sbiofit(_,Name,Value)

[fitResults,simdata] = sbiofit(_)
```

### Description

`fitResults = sbiofit(sm,grpData,responseMap,estiminfo)` estimates parameters of a SimBiology model `sm` using nonlinear least-squares regression.

`grpData` is a `groupedData` object specifying the data to fit. `responseMap` defines the mapping between the model components and response data in `grpData`.

`estimatedInfo` is an `estimatedInfo` object that defines the estimated parameters in the model `sm`. `fitResults` is a `OptimResults` object or `NLINResults` object or a vector of these objects.

`sbiofit` uses the first available estimation function among the following: `lsqnonlin` (Optimization Toolbox required), `nlinfit` (Statistics and Machine Learning Toolbox required), or `fminsearch`.

By default, each group in `grpData` is fit separately, resulting in group-specific parameter estimates. If the model contains active doses and variants, they are applied before the simulation.

`fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing)` uses the dosing information specified by a matrix of SimBiology dose objects `dosing` instead of using the active doses of the model `sm` if there is any.

`fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing,functionName)` uses the estimation function specified by `functionName`. If the specified function is unavailable, a warning is issued and the first available default function is used.

`fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing,functionName,options)` uses the additional options specified by `options` for the function `functionName`.

`fitResults = sbiofit(sm,grpData,responseMap,estiminfo,dosing,functionName,options,variants)` applies variant objects specified as `variants` instead of using any active variants of the model.

`fitResults = sbiofit(_,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

`[fitResults,simdata] = sbiofit(_)` also returns a vector of SimData objects `simdata` using any of the input arguments in the previous syntaxes.

---

**Note**

- `sbiofit` unifies `sbionlinfit` and `sbioparamestim` estimation functions. Use `sbiofit` to perform nonlinear least-squares regression.
  - `sbiofit` simulates the model using a `SimFunction` object, which automatically accelerates simulations by default. Hence it is not necessary to run `sbioaccelerate` before you call `sbiofit`.
- 

## Examples

## Fit a One-Compartment Model to an Individual's PK Profile

### Background

This example shows how to fit an individual's PK profile data to one-compartment model and estimate pharmacokinetic parameters.

Suppose you have drug plasma concentration data from an individual and want to estimate the volume of the central compartment and the clearance. Assume the drug

concentration versus the time profile follows the monoexponential decline  $C_t = C_0 e^{-k_e t}$ , where  $C_t$  is the drug concentration at time  $t$ ,  $C_0$  is the initial concentration, and  $k_e$  is the elimination rate constant that depends on the clearance and volume of the central compartment  $k_e = Cl/V$ .

The synthetic data in this example was generated using the following model and parameters:

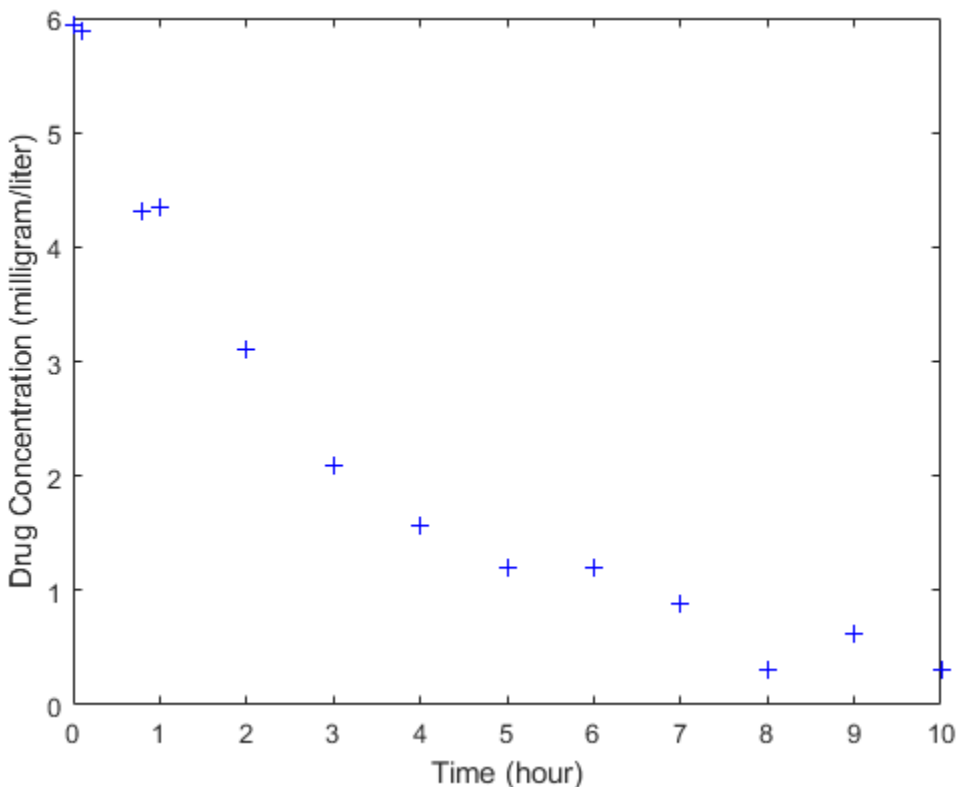
- One-compartment model with bolus dosing and first-order elimination
- Volume of the central compartment (Central) = 1.70 liter
- Clearance parameter (Cl\_Central) = 0.55 liter/hour
- Constant error model

### Load Data and Visualize

The data is stored as a table with variables Time and Conc that represent the time course of the plasma concentration of an individual after an intravenous bolus administration measured at 13 different time points. The variable units for Time and Conc are hour and milligram/liter, respectively.

```
clear all
load(fullfile(matlabroot, 'examples', 'simbio', 'data15.mat'))

plot(data.Time, data.Conc, 'b+')
xlabel('Time (hour)');
ylabel('Drug Concentration (milligram/liter)');
```



### Convert to groupedData Format

Convert the data set to a `groupedData` object, which is the required data format for the fitting function `sbiofit` for later use. A `groupedData` object also lets you set independent variable and group variable names (if they exist). Set the units of the `Time` and `Conc` variables. The units are optional and only required for the `UnitConversion` feature, which automatically converts matching physical quantities to one consistent unit system.

```
gData = groupedData(data);  
gData.Properties.VariableUnits = {'hour', 'milligram/liter'};  
gData.Properties
```

```
ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Row' 'Variables'}
    VariableNames: {'Time' 'Conc'}
    VariableDescriptions: {}
    VariableUnits: {'hour' 'milligram/liter'}
    VariableContinuity: []
    RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: ''
    IndependentVariableName: 'Time'
```

groupedData automatically set the name of the IndependentVariableName property to the Time variable of the data.

## Construct a One-Compartment Model

Use the built-in PK library to construct a one-compartment model with bolus dosing and first-order elimination where the elimination rate depends on the clearance and volume of the central compartment. Use the configset object to turn on unit conversion.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
model = construct(pkmd);
configset = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

For details on creating compartmental PK models using the SimBiology® built-in library, see “Create Pharmacokinetic Models”.

## Define Dosing

Define a single bolus dose of 10 milligram given at time = 0. For details on setting up different dosing schedules, see “Doses”.

```
dose = sbiodose('dose');
dose.TargetName = 'Drug_Central';
dose.StartTime = 0;
dose.Amount = 10;
```

```
dose.AmountUnits = 'milligram';
dose.TimeUnits   = 'hour';
```

### Map Response Data to the Corresponding Model Component

The data contains drug concentration data stored in the `Conc` variable. This data corresponds to the `Drug_Central` species in the model. Therefore, map the data to `Drug_Central` as follows.

```
responseMap = {'Drug_Central = Conc'};
```

### Specify Parameters to Estimate

The parameters to fit in this model are the volume of the central compartment (`Central`) and the clearance rate (`Cl_Central`). In this case, specify log-transformation for these biological parameters since they are constrained to be positive. The `estimatedInfo` object lets you specify parameter transforms, initial values, and parameter bounds if needed.

```
paramsToEstimate = {'log(Central)', 'log(Cl_Central)'};
estimatedParams  = estimatedInfo(paramsToEstimate, 'InitialValue', [1 1], 'Bounds', [1 5]);
```

### Estimate Parameters

Now that you have defined one-compartment model, data to fit, mapped response data, parameters to estimate, and dosing, use `sbiofit` to estimate parameters. The default estimation function that `sbiofit` uses will change depending on which toolboxes are available. To see which function was used during fitting, check the `EstimationFunction` property of the corresponding results object.

```
fitConst = sbiofit(model, gData, responseMap, estimatedParams, dose);
```

### Display Estimated Parameters and Plot Results

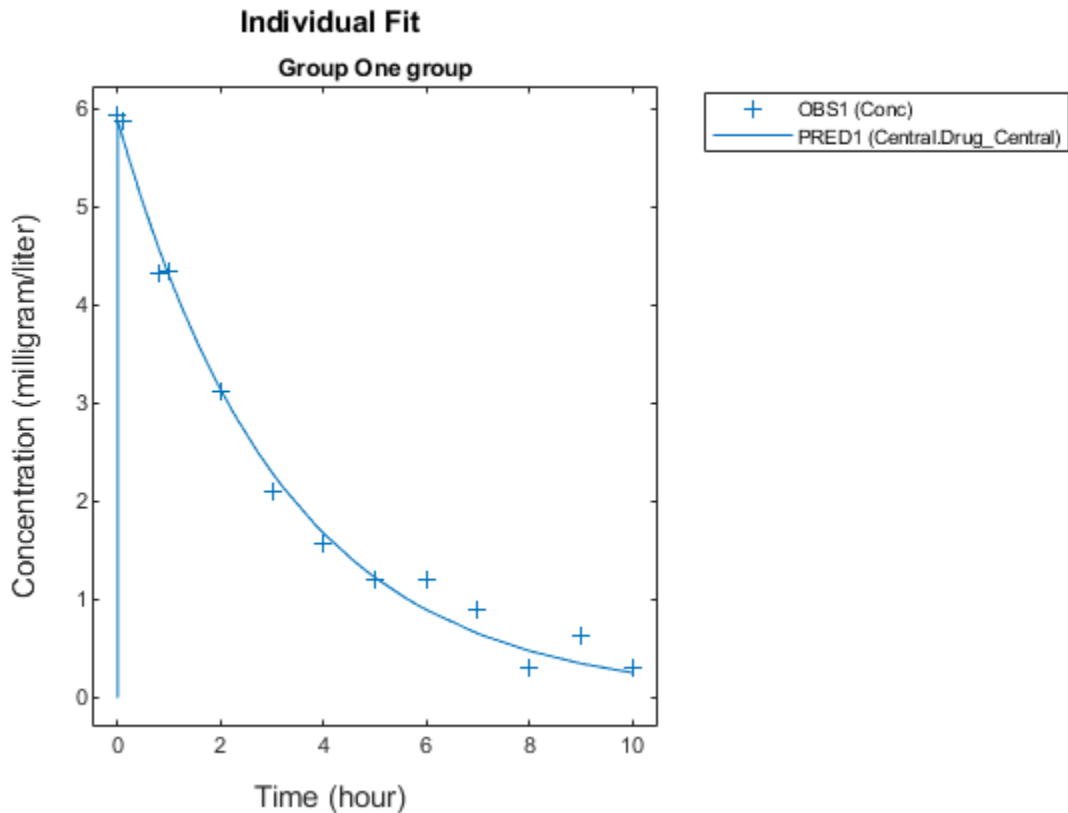
Notice the parameter estimates were not far off from the true values (1.70 and 0.55) that were used to generate the data. You may also try different error models to see if they could further improve the parameter estimates.

```
fitConst.ParameterEstimates
```

```
ans=2x4 table
      Name      Estimate      StandardError      Bounds
```

```
'Central'      1.6993      0.034821      1      5
'Cl_Central'   0.53358     0.01968      0.5     2
```

```
s.Labels.XLabel = 'Time (hour)';
s.Labels.YLabel = 'Concentration (milligram/liter)';
plot(fitConst, 'AxesStyle', s);
```



### Use Different Error Models

Try three other supported error models (proportional, combination of constant and proportional error models, and exponential).

```
fitProp = sbiofit(model, gData, responseMap, estimatedParams, dose, ...
    'ErrorModel', 'proportional');
```



```

fitExp = sbiofit(model,gData,responseMap,estimatedParams,dose,...
               'ErrorModel','exponential');
fitComb = sbiofit(model,gData,responseMap,estimatedParams,dose,...
                 'ErrorModel','combined');

```

### Use Weights Instead of an Error Model

You can specify weights as a numeric matrix, where the number of columns corresponds to the number of responses. Setting all weights to 1 is equivalent to the constant error model.

```

weightsNumeric = ones(size(gData.Conc));
fitWeightsNumeric = sbiofit(model,gData,responseMap,estimatedParams,dose,'Weights',wei

```

Alternatively, you can use a function handle that accepts a vector of predicted response values and returns a vector of weights. In this example, use a function handle that is equivalent to the proportional error model.

```

weightsFunction = @(y) 1./y.^2;
fitWeightsFunction = sbiofit(model,gData,responseMap,estimatedParams,dose,'Weights',wei

```

### Compare Information Criteria for Model Selection

Compare the loglikelihood, AIC, and BIC values of each model to see which error model best fits the data. A larger likelihood value indicates the corresponding model fits the model better. For AIC and BIC, the smaller values are better.

```

allResults = [fitConst,fitWeightsNumeric,fitWeightsFunction,fitProp,fitExp,fitComb];
errorModelNames = {'constant error model','equal weights','proportional weights', ...
                  'proportional error model','exponential error model',...
                  'combined error model'};
LogLikelihood = [allResults.LogLikelihood]';
AIC = [allResults.AIC]';
BIC = [allResults.BIC]';
t = table(LogLikelihood,AIC,BIC);
t.Properties.RowNames = errorModelNames;
t

```

t=6x3 table

	LogLikelihood	AIC	BIC
constant error model	3.9866	-3.9732	-2.8433
equal weights	3.9866	-3.9732	-2.8433
proportional weights	-3.8472	11.694	12.824

proportional error model	-3.8257	11.651	12.781
exponential error model	1.1984	1.6032	2.7331
combined error model	3.9163	-3.8326	-2.7027

Based on the information criteria, the constant error model (or equal weights) fits the data best since it has the largest loglikelihood value and the smallest AIC and BIC.

### Display Estimated Parameter Values

Show the estimated parameter values of each model.

```
Estimated_Central      = zeros(6,1);
Estimated_Cl_Central   = zeros(6,1);
t2 = table(Estimated_Central,Estimated_Cl_Central);
t2.Properties.RowNames = errorModelNames;
for i = 1:height(t2)
    t2{i,1} = allResults(i).ParameterEstimates.Estimate(1);
    t2{i,2} = allResults(i).ParameterEstimates.Estimate(2);
end
t2
```

t2=6×2 table

	<u>Estimated_Central</u>	<u>Estimated_Cl_Central</u>
constant error model	1.6993	0.53358
equal weights	1.6993	0.53358
proportional weights	1.9045	0.51734
proportional error model	1.8777	0.51147
exponential error model	1.7872	0.51701
combined error model	1.7008	0.53271

### Conclusion

This example showed how to estimate PK parameters, namely the volume of the central compartment and clearance parameter of an individual, by fitting the PK profile data to one-compartment model. You compared the information criteria of each model and estimated parameter values of different error models to see which model best explained the data. Final fitted results suggested both the constant and combined error models provided the closest estimates to the parameter values used to generate the data.

However, the constant error model is a better model as indicated by the loglikelihood, AIC, and BIC information criteria.

### Fit a Two-Compartment Model to PK Profiles of Multiple Individuals

Suppose you have drug plasma concentration data from three individuals that you want to use to estimate corresponding pharmacokinetic parameters, namely the volume of central and peripheral compartment (`Central`, `Peripheral`), the clearance rate (`CL_Central`), and intercompartmental clearance (`Q12`). Assume the drug concentration versus the time

profile follows the biexponential decline  $C_t = Ae^{-at} + Be^{-bt}$ , where  $C_t$  is the drug concentration at time  $t$ , and  $a$  and  $b$  are slopes for corresponding exponential declines.

The synthetic data set contains drug plasma concentration data measured in both central and peripheral compartments. The data was generated using a two-compartment model with an infusion dose and first-order elimination. These parameters were used for each individual.

	Central	Peripheral	Q12	CL_Central
Individual 1	1.90	0.68	0.24	0.57
Individual 2	2.10	6.05	0.36	0.95
Individual 3	1.70	4.21	0.46	0.95

The data is stored as a table with variables `ID`, `Time`, `CentralConc`, and `PeripheralConc`. It represents the time course of plasma concentrations measured at eight different time points for both central and peripheral compartments after an infusion dose.

```
clear all
load(fullfile(matlabroot, 'examples', 'simbio', 'data10_32R.mat'))
```

Convert the data set to a `groupedData` object which is the required data format for the fitting function `sbiofit` for later use. A `groupedData` object also lets you set independent variable and group variable names (if they exist). Set the units of the `ID`, `Time`, `CentralConc`, and `PeripheralConc` variables. The units are optional and only required for the `UnitConversion` feature, which automatically converts matching physical quantities to one consistent unit system.

```
gData = groupedData(data);  
gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};  
gData.Properties
```

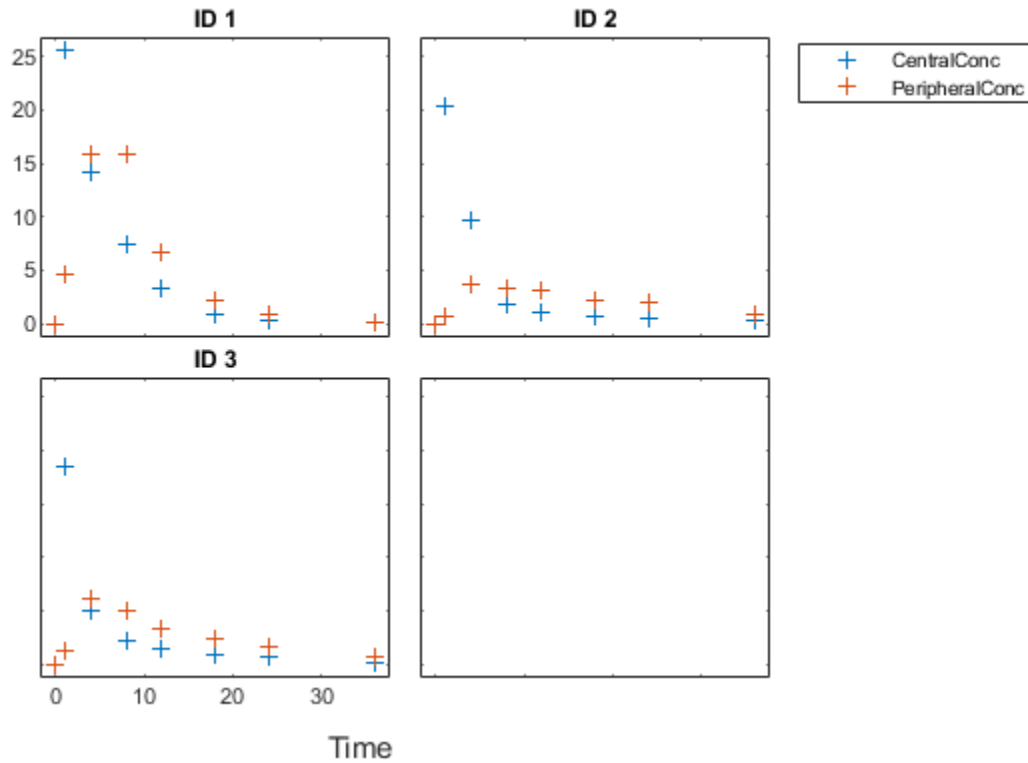
```
ans =
```

```
struct with fields:
```

```
    Description: ''  
    UserData: []  
    DimensionNames: {'Row' 'Variables'}  
    VariableNames: {'ID' 'Time' 'CentralConc' 'PeripheralConc'}  
    VariableDescriptions: {}  
    VariableUnits: {1x4 cell}  
    VariableContinuity: []  
    RowNames: {}  
    CustomProperties: [1x1 matlab.tabular.CustomProperties]  
    GroupVariableName: 'ID'  
    IndependentVariableName: 'Time'
```

Create a trellis plot that shows the PK profiles of three individuals.

```
sbiotrellis(gData, 'ID', 'Time', {'CentralConc', 'PeripheralConc'}, ...  
            'Marker', '+', 'LineStyle', 'none');
```



Use the built-in PK library to construct a two-compartment model with infusion dosing and first-order elimination where the elimination rate depends on the clearance and volume of the central compartment. Use the configset object to turn on unit conversion.

```
pkmd          = PKModelDesign;
pkc1          = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2          = addCompartment(pkmd, 'Peripheral');
model        = construct(pkmd);
configset    = getconfigset(model);
configset.CompileOptions.UnitConversion = true;
```

Assume every individual receives an infusion dose at time = 0, with a total infusion amount of 100 mg at a rate of 50 mg/hour. For details on setting up different dosing strategies, see “Doses”.

```
dose                = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime     = 0;
dose.Amount        = 100;
dose.Rate          = 50;
dose.AmountUnits   = 'milligram';
dose.TimeUnits     = 'hour';
dose.RateUnits     = 'milligram/hour';
```

The data contains measured plasma concentrations in the central and peripheral compartments. Map these variables to the appropriate model species, which are `Drug_Central` and `Drug_Peripheral`.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
```

The parameters to estimate in this model are the volumes of central and peripheral compartments (`Central` and `Peripheral`), intercompartmental clearance `Q12`, and clearance rate `Cl_Central`. In this case, specify log-transform for `Central` and `Peripheral` since they are constrained to be positive. The `estimatedInfo` object lets you specify parameter transforms, initial values, and parameter bounds (optional).

```
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam   = estimatedInfo(paramsToEstimate, 'InitialValue', [1 1 1 1]);
```

Fit the model to all of the data pooled together, that is, estimate one set of parameters for all individuals. The default estimation method that `sbiofit` uses will change depending on which toolboxes are available. To see which estimation function `sbiofit` used for the fitting, check the `EstimationFunction` property of the corresponding results object.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true)
```

```
pooledFit =
```

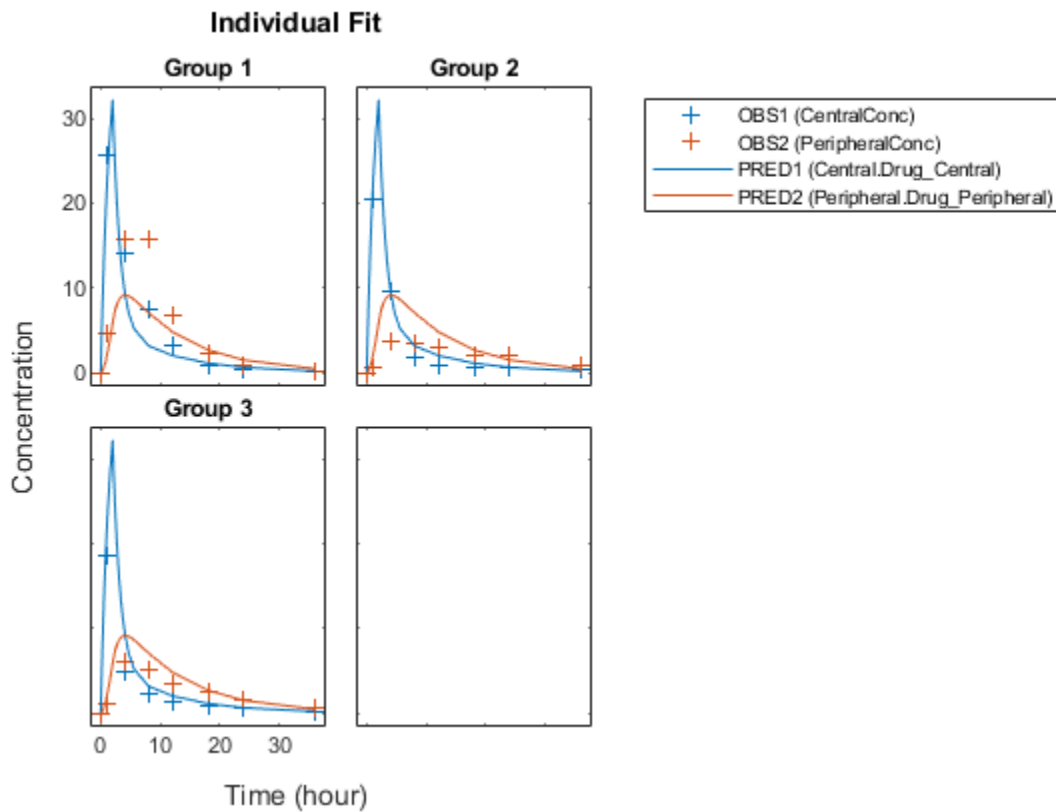
```
  OptimResults with properties:
```

```
      ExitFlag: 3
      Output: [1x1 struct]
      GroupName: []
      Beta: [4x3 table]
      ParameterEstimates: [4x3 table]
```

```
J: [24x4x2 double]
COVB: [4x4 double]
CovarianceMatrix: [4x4 double]
R: [24x2 double]
MSE: 6.6220
SSE: 291.3688
Weights: []
LogLikelihood: -111.3904
AIC: 230.7808
BIC: 238.2656
DFE: 44
DependentFiles: {1x3 cell}
EstimatedParameterNames: {'Central' 'Peripheral' 'Q12' 'Cl_Central'}
ErrorModelInfo: [1x3 table]
EstimationFunction: 'lsqnonlin'
```

Plot the fitted results versus the original data. Although three separate plots were generated, the data was fitted using the same set of parameters (that is, all three individuals had the same fitted line).

```
plot(pooledFit);
```



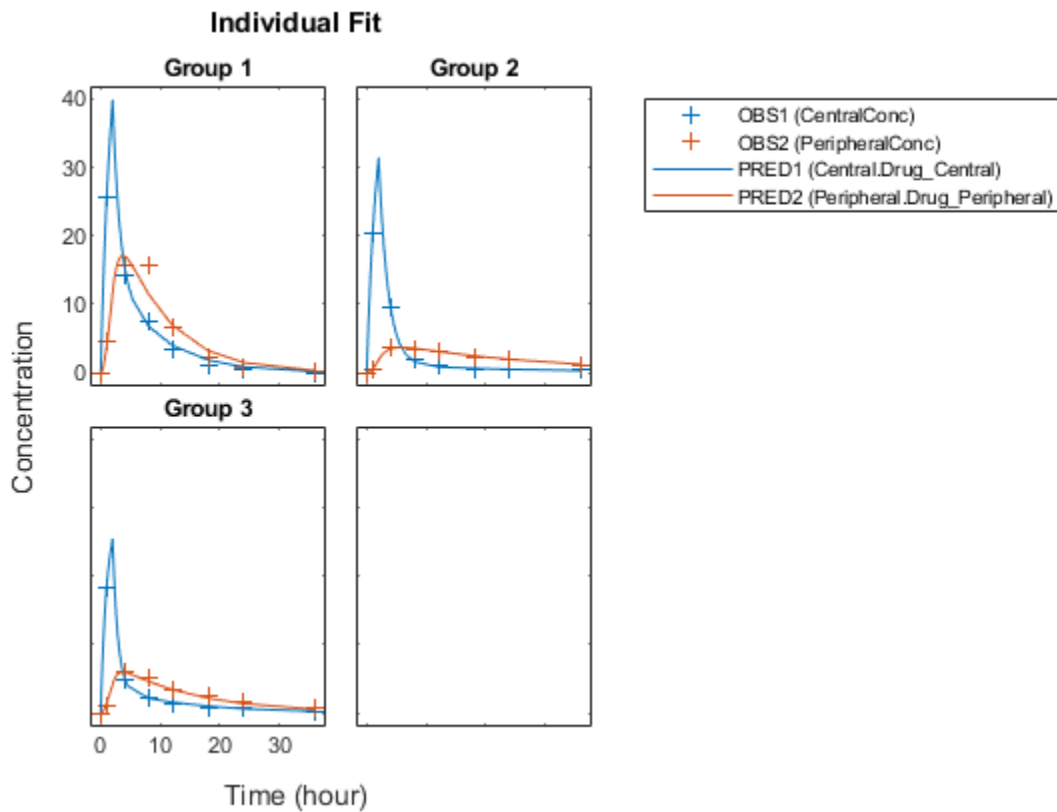
Estimate one set of parameters for each individual and see if there is any improvement in the parameter estimates. In this example, since there are three individuals, three sets of parameters are estimated.

```
unpooledFit = sbiofit(model,gData,responseMap,estimatedParam,dose,'Pooled',false);
```

Plot the fitted results versus the original data. Each individual was fitted differently (that is, each fitted line is unique to each individual) and each line appeared to fit well to individual data.

```
plot(unpooledFit);
```





Display the fitted results of the first individual. The MSE was lower than that of the pooled fit. This is also true for the other two individuals.

```
unpooledFit(1)
```

```
ans =
```

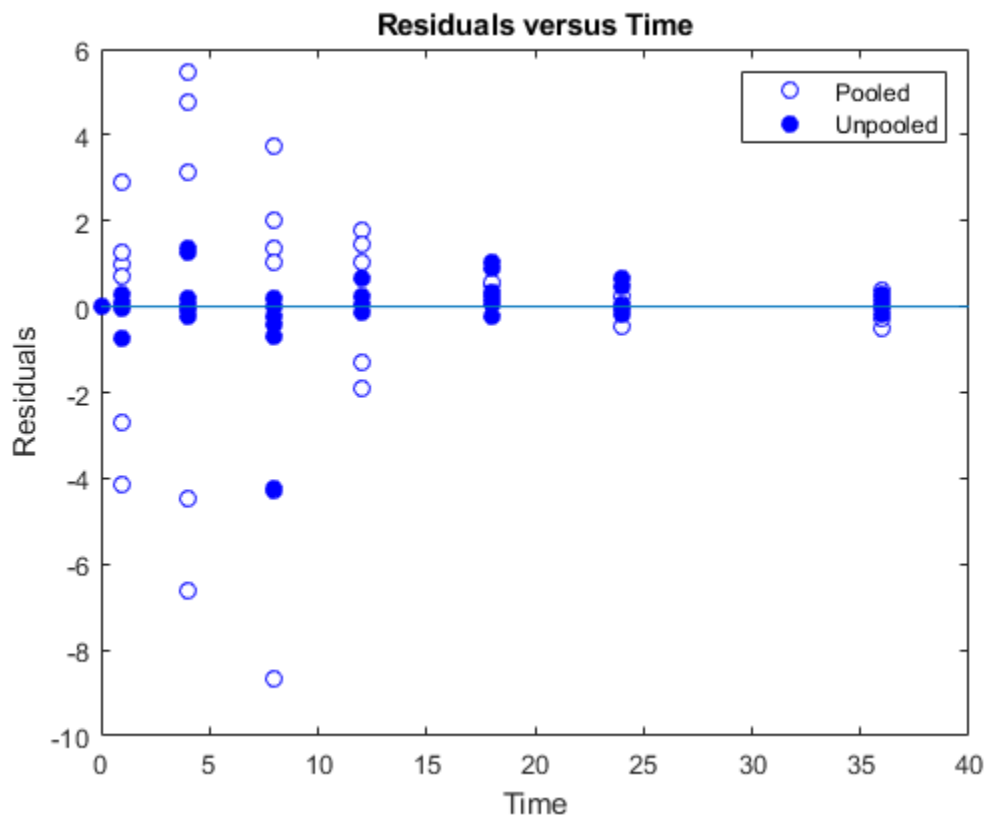
```
OptimResults with properties:
```

```
ExitFlag: 3
Output: [1x1 struct]
GroupName: 1
Beta: [4x3 table]
```

```
ParameterEstimates: [4x3 table]
                   J: [8x4x2 double]
                   COVB: [4x4 double]
CovarianceMatrix: [4x4 double]
                   R: [8x2 double]
                   MSE: 2.1380
                   SSE: 25.6559
                   Weights: []
LogLikelihood: -26.4805
                   AIC: 60.9610
                   BIC: 64.0514
                   DFE: 12
DependentFiles: {1x3 cell}
EstimatedParameterNames: {'Central' 'Peripheral' 'Q12' 'Cl_Central'}
ErrorModelInfo: [1x3 table]
EstimationFunction: 'lsqnonlin'
```

Generate a plot of the residuals over time to compare the pooled and unpooled fit results. The figure indicates unpooled fit residuals are smaller than those of pooled fit as expected. In addition to comparing residuals, other rigorous criteria can be used to compare the fitted results.

```
t = [gData.Time;gData.Time];
res_pooled = vertcat(pooledFit.R);
res_pooled = res_pooled(:);
res_unpooled = vertcat(unpooledFit.R);
res_unpooled = res_unpooled(:);
plot(t,res_pooled,'o','MarkerFaceColor','w','markerEdgeColor','b')
hold on
plot(t,res_unpooled,'o','MarkerFaceColor','b','markerEdgeColor','b')
refl = refline(0,0); % A reference line representing a zero residual
title('Residuals versus Time');
xlabel('Time');
ylabel('Residuals');
legend({'Pooled','Unpooled'});
```



This example showed how to perform pooled and unpooled estimations using `sbiofit`. As illustrated, the unpooled fit accounts for variations due to the specific subjects in the study, and, in this case, the model fits better to the data. However, the pooled fit returns population-wide parameters. If you want to estimate population-wide parameters while considering individual variations, use `sbiofitmixed`.

### Estimate Category-Specific PK Parameters for Multiple Individuals

This example shows how to estimate category-specific (such as young versus old, male versus female), individual-specific, and population-wide parameters using PK profile data from multiple individuals.

## Background

Suppose you have drug plasma concentration data from 30 individuals and want to estimate pharmacokinetic parameters, namely the volumes of central and peripheral compartment, the clearance, and intercompartmental clearance. Assume the drug concentration versus the time profile follows the biexponential decline

$C_t = Ae^{-at} + Be^{-bt}$ , where  $C_t$  is the drug concentration at time  $t$ , and  $a$  and  $b$  are slopes for corresponding exponential declines.

## Load Data

This synthetic data contains the time course of plasma concentrations of 30 individuals after a bolus dose (100 mg) measured at different times for both central and peripheral compartments. It also contains categorical variables, namely Sex and Age.

```
clear
load(fullfile(matlabroot, 'examples', 'simbio', 'sd5_302RAgeSex.mat'))
```

## Convert to groupedData Format

Convert the data set to a `groupedData` object, which is the required data format for the fitting function `sbiofit`. A `groupedData` object also allows you set independent variable and group variable names (if they exist). Set the units of the ID, Time, CentralConc, PeripheralConc, Age, and Sex variables. The units are optional and only required for the `UnitConversion` feature, which automatically converts matching physical quantities to one consistent unit system.

```
gData = groupedData(data);
gData.Properties.VariableUnits = {'', 'hour', 'milligram/liter', 'milligram/liter', '', ''};
gData.Properties
```

```
ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Row' 'Variables'}
    VariableNames: {1x6 cell}
    VariableDescriptions: {}
    VariableUnits: {1x6 cell}
    VariableContinuity: []
    RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: 'ID'
```

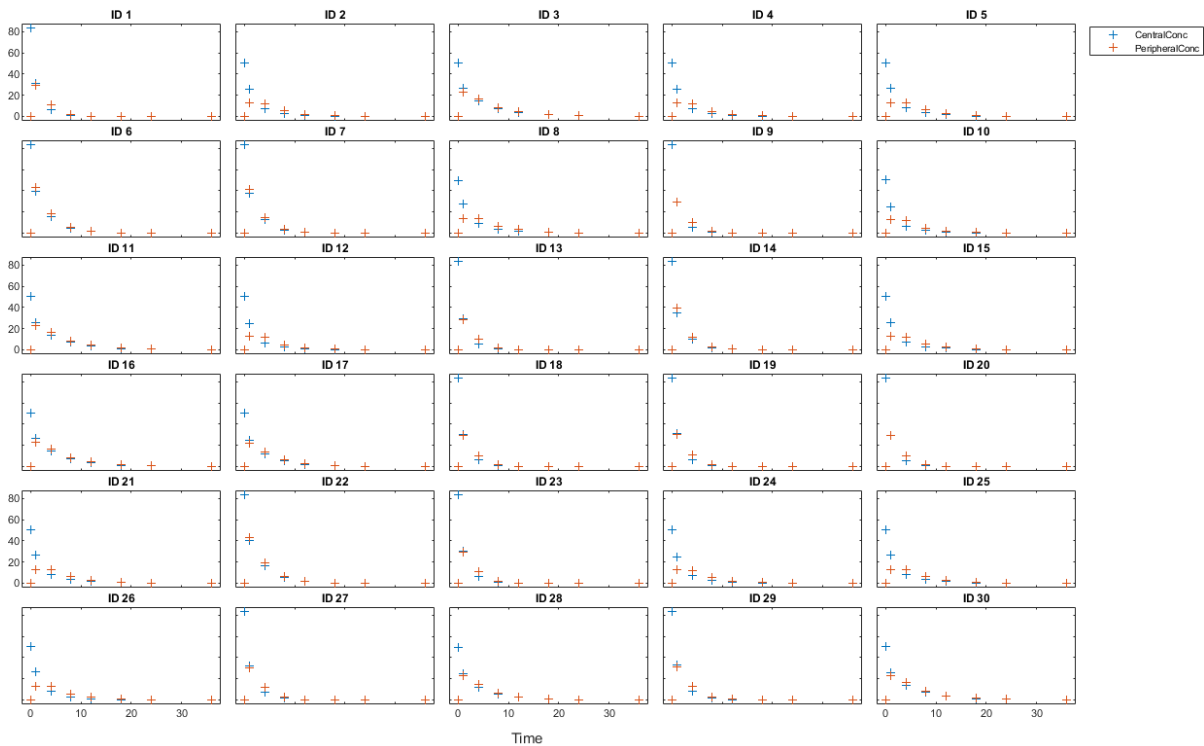
```
IndependentVariableName: 'Time'
```

The `IndependentVariableName` and `GroupVariableName` properties have been automatically set to the `Time` and `ID` variables of the data.

## Visualize Data

Display the response data for each individual.

```
t = sbiotrellis(gData, 'ID', 'Time', {'CentralConc', 'PeripheralConc'}, ...
    'Marker', '+', 'LineStyle', 'none');
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```



## Set Up a Two-Compartment Model

Use the built-in PK library to construct a two-compartment model with infusion dosing and first-order elimination where the elimination rate depends on the clearance and volume of the central compartment. Use the `configset` object to turn on unit conversion.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2 = addCompartment(pkmd, 'Peripheral');
model = construct(pkmd);
configset = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

For details on creating compartmental PK models using the SimBiology® built-in library, see “Create Pharmacokinetic Models”.

## Define Dosing

Assume every individual receives a bolus dose of 100 mg at time = 0. For details on setting up different dosing strategies, see “Doses”.

```
dose = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime = 0;
dose.Amount = 100;
dose.AmountUnits = 'milligram';
dose.TimeUnits = 'hour';
```

## Map the Response Data to Corresponding Model Components

The data contains measured plasma concentration in the central and peripheral compartments. Map these variables to the appropriate model components, which are `Drug_Central` and `Drug_Peripheral`.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
```

## Specify Parameters to Estimate

Specify the volumes of central and peripheral compartments `Central` and `Peripheral`, intercompartmental clearance `Q12`, and clearance `Cl_Central` as parameters to estimate. The `estimatedInfo` object lets you optionally specify parameter transforms,

initial values, and parameter bounds. Since both `Central` and `Peripheral` are constrained to be positive, specify a log-transform for each parameter.

```
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};  
estimatedParam   = estimatedInfo(paramsToEstimate, 'InitialValue', [1 1 1 1]);
```

### Estimate Individual-Specific Parameters

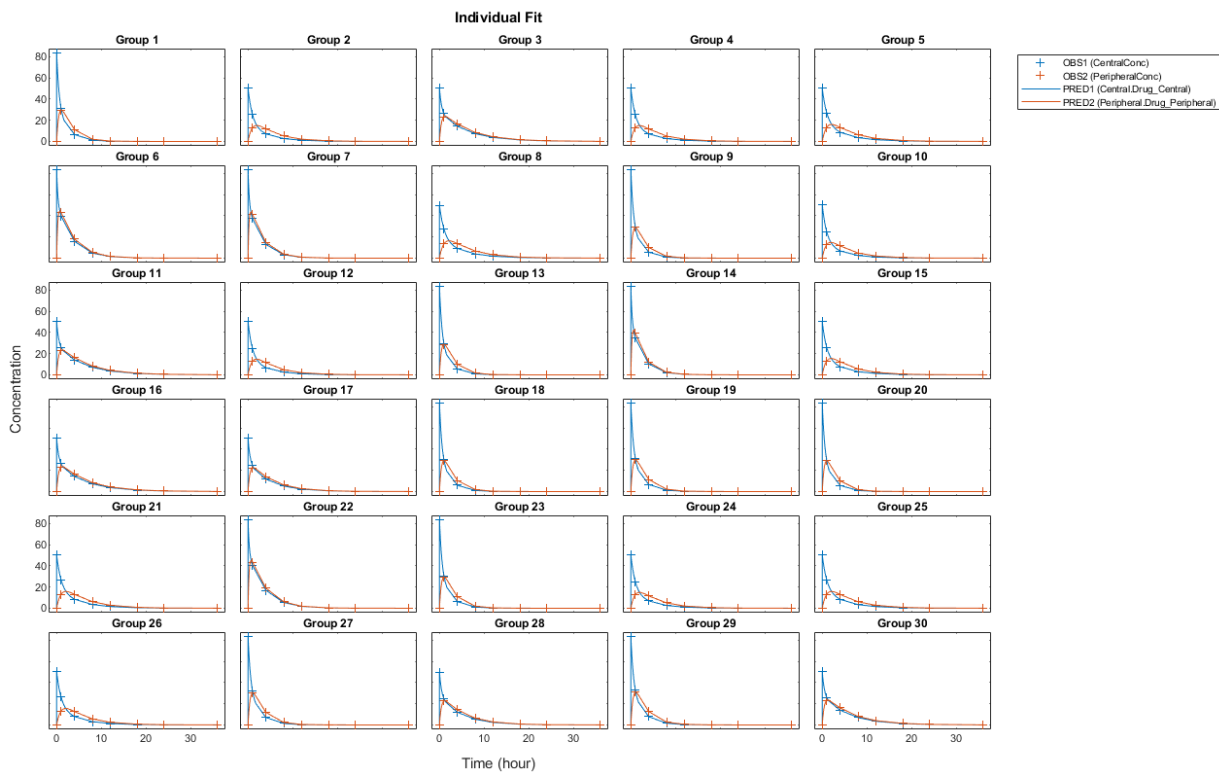
Estimate one set of parameters for each individual by setting the `'Pooled'` name-value pair argument to `false`.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

### Display Results

Plot the fitted results versus the original data for each individual (group).

```
t = plot(unpooledFit);  
% Resize the figure.  
t.hFig.Position(:) = [100 100 1280 800];
```



For an unpooled fit, `sbiofit` always returns one results object for each individual.

## Examine Parameter Estimates for Category Dependencies

Explore the unpooled estimates to see if there is any category-specific parameters, that is, if some parameters are related to one or more categories. If there are any category dependencies, it might be possible to reduce the number of degrees of freedom by estimating just category-specific values for those parameters.

First extract the ID and category values for each ID

```
catParamValues = unique(gData(:, {'ID', 'Sex', 'Age'}));
```

Add variables to the table containing each parameter's estimate.

```
allParamValues = vertcat(unpooledFit.ParameterEstimates);
catParamValues.Central = allParamValues.Estimate(strcmp(allParamValues.Name, 'Centra
```



```

catParamValues.Peripheral = allParamValues.Estimate(strcmp(allParamValues.Name, 'Peripheral'), allParamValues.Estimate);
catParamValues.Q12       = allParamValues.Estimate(strcmp(allParamValues.Name, 'Q12'), allParamValues.Estimate);
catParamValues.Cl_Central = allParamValues.Estimate(strcmp(allParamValues.Name, 'Cl_Central'), allParamValues.Estimate);

```

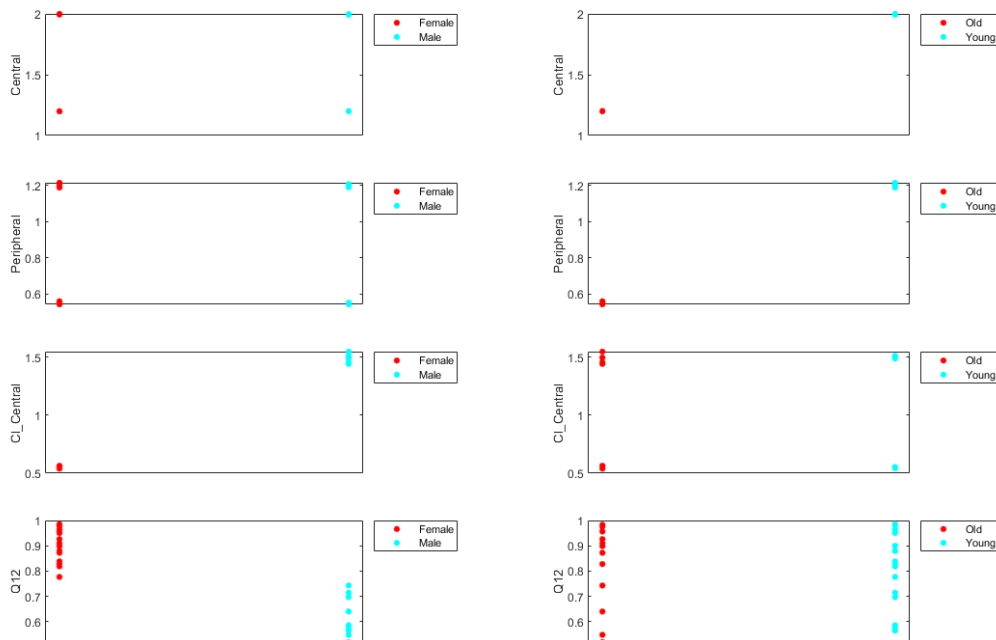
Plot estimates of each parameter for each category. `gscatter` requires Statistics and Machine Learning Toolbox™. If you do not have it, use other alternative plotting functions such as `plot`.

```

h          = figure;
ylabel     = {'Central', 'Peripheral', 'Cl_Central', 'Q12'};
plotNumber = 1;
for i = 1:4
    thisParam = estimatedParam(i).Name;

    % Plot for Sex category
    subplot(4,2,plotNumber);
    plotNumber = plotNumber + 1;
    gscatter(double(catParamValues.Sex), catParamValues.(thisParam), catParamValues.Sex);
    ax        = gca;
    ax.XTick  = [];
    ylabel(ylabels(i));
    legend('Location', 'bestoutside')
    % Plot for Age category
    subplot(4,2,plotNumber);
    plotNumber = plotNumber + 1;
    gscatter(double(catParamValues.Age), catParamValues.(thisParam), catParamValues.Age);
    ax        = gca;
    ax.XTick  = [];
    ylabel(ylabels(i));
    legend('Location', 'bestoutside')
end
% Resize the figure.
h.Position(:) = [100 100 1280 800];

```



Based on the plot, it seems that young individuals tend to have higher volumes of central and peripheral compartments (Central, Peripheral) than old individuals (that is, the volumes seem to be age-specific). In addition, males tend to have lower clearance rates (Cl\_Central) than females but the opposite for the Q12 parameter (that is, the clearance and Q12 seem to be sex-specific).

## Estimate Category-Specific Parameters

Use the 'CategoryVariableName' property of the estimatedInfo object to specify which category to use during fitting. Use 'Sex' as the group to fit for the clearance Cl\_Central and Q12 parameters. Use 'Age' as the group to fit for the Central and Peripheral parameters.

```
estimatedParam(1).CategoryVariableName = 'Age';
estimatedParam(2).CategoryVariableName = 'Age';
estimatedParam(3).CategoryVariableName = 'Sex';
```

```

estimatedParam(4).CategoryVariableName = 'Sex';
categoryFit = sbiofit(model,gData,responseMap,estimatedParam,dose)

categoryFit =
  OptimResults with properties:

      ExitFlag: 3
      Output: [1x1 struct]
      GroupName: []
      Beta: [8x5 table]
      ParameterEstimates: [120x6 table]
      J: [240x8x2 double]
      COVB: [8x8 double]
      CovarianceMatrix: [8x8 double]
      R: [240x2 double]
      MSE: 0.4362
      SSE: 205.8690
      Weights: []
      LogLikelihood: -477.9195
      AIC: 971.8390
      BIC: 1.0052e+03
      DFE: 472
      DependentFiles: {1x3 cell}
      EstimatedParameterNames: {'Central' 'Peripheral' 'Q12' 'Cl_Central'}
      ErrorModelInfo: [1x3 table]
      EstimationFunction: 'lsqnonlin'

```

When fitting by category (or group), `sbiofit` always returns one results object, not one for each category level. This is because both male and female individuals are considered to be part of the same optimization using the same error model and error parameters, similarly for the young and old individuals.

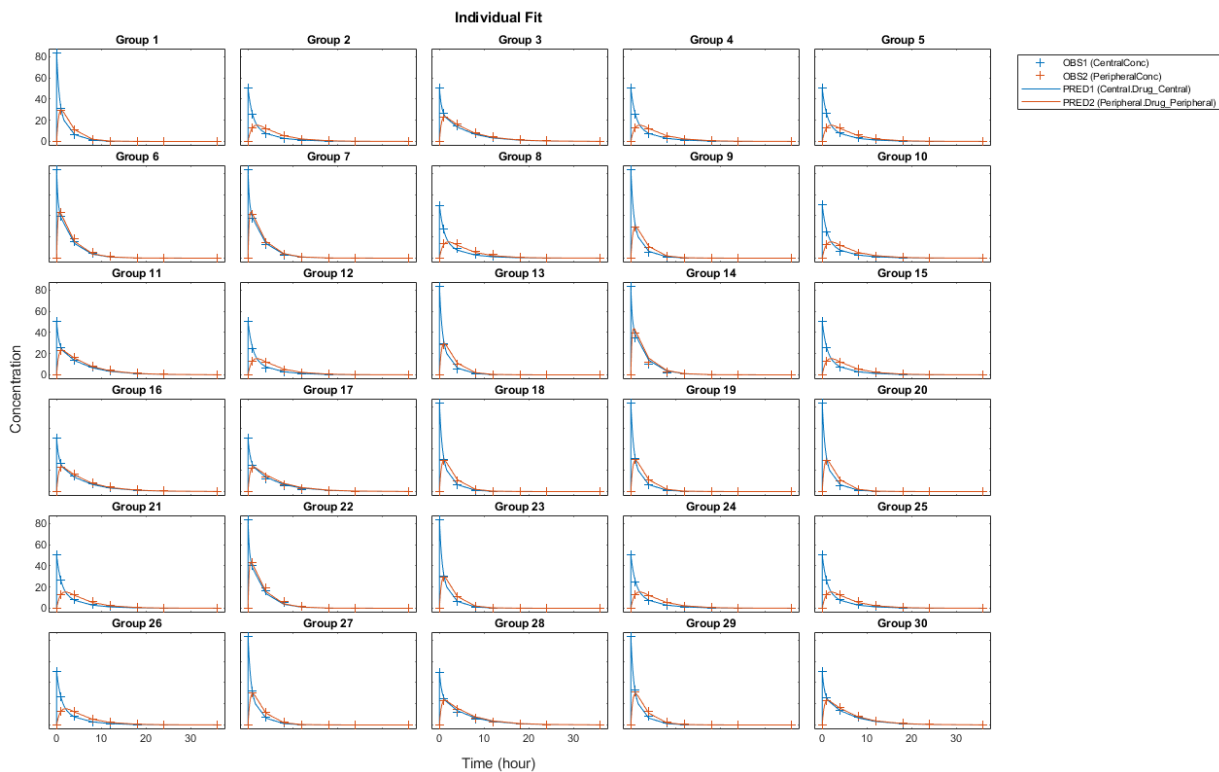
### Plot Results

Plot the category-specific estimated results.

```

t = plot(categoryFit);
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];

```



For the `Cl_Central` and `Q12` parameters, all males had the same estimates, and similarly for the females. For the `Central` and `Peripheral` parameters, all young individuals had the same estimates, and similarly for the old individuals.

## Estimate Population-Wide Parameters

To better compare the results, fit the model to all of the data pooled together, that is, estimate one set of parameters for all individuals by setting the `'Pooled'` name-value pair argument to `true`. The warning message tells you that this option will ignore any category-specific information (if they exist).

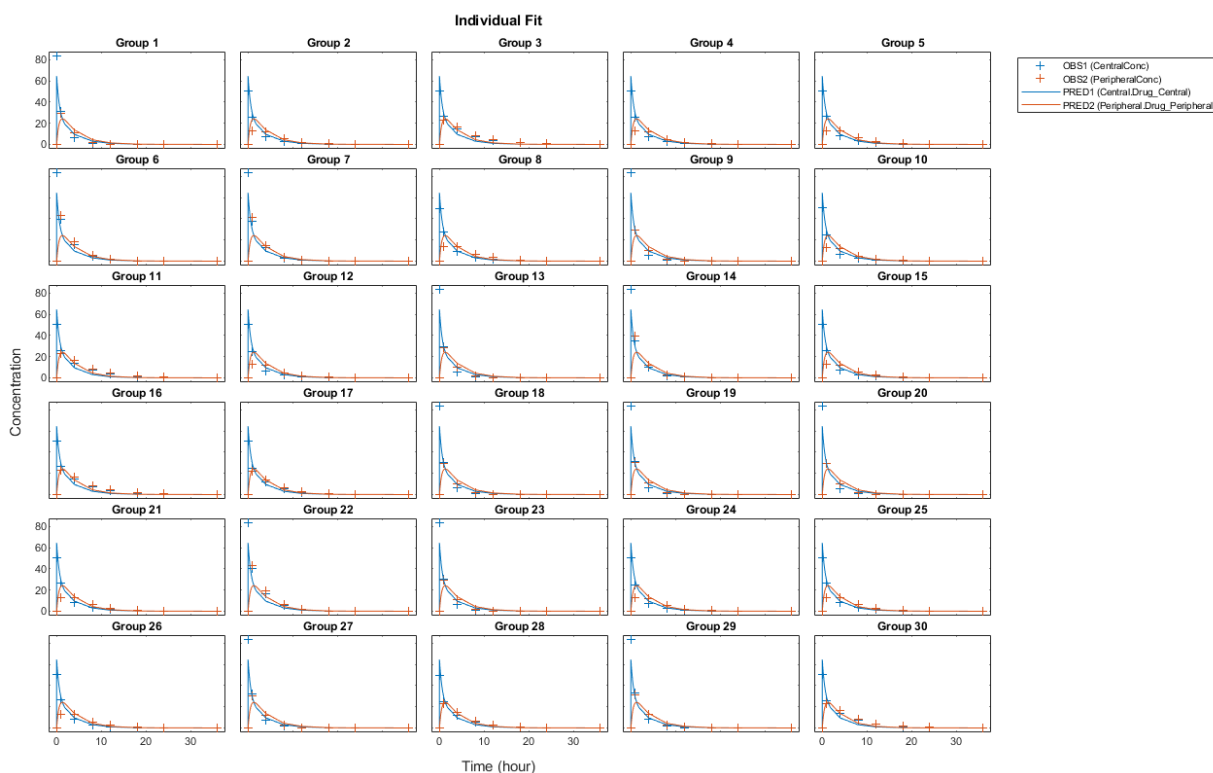
```
pooledFit = sbiofit(model,gData,responseMap,estimatedParam,dose,'Pooled',true);
```

```
Warning: CategoryVariableName property of the estimatedInfo object is ignored when using
```

## Plot Results

Plot the fitted results versus the original data. Although a separate plot was generated for each individual, the data was fitted using the same set of parameters (that is, all individuals had the same fitted line).

```
t = plot(pooledFit);
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```



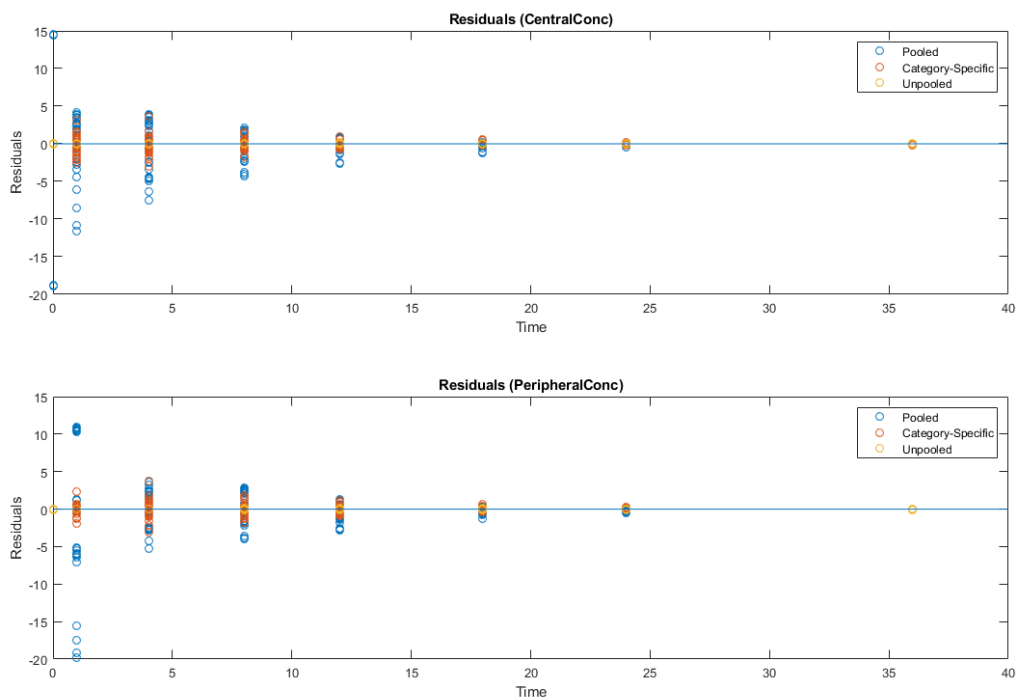
## Compare Residuals

Compare residuals of CentralConc and PeripheralConc responses for each fit.

```
t = gData.Time;
allResid(:, :, 1) = pooledFit.R;
```

```
allResid(:,:,2) = categoryFit.R;
allResid(:,:,3) = vertcat(unpooledFit.R);

h = figure;
responseList = {'CentralConc', 'PeripheralConc'};
for i = 1:2
    subplot(2,1,i);
    oneResid = squeeze(allResid(:,i,:));
    plot(t,oneResid,'o');
    reffline(0,0); % A reference line representing a zero residual
    title(sprintf('Residuals (%s)', responseList{i}));
    xlabel('Time');
    ylabel('Residuals');
    legend({'Pooled', 'Category-Specific', 'Unpooled'});
end
% Resize the figure.
h.Position(:) = [100 100 1280 800];
```



As shown in the plot, the unpooled fit produced the best fit to the data as it fit the data to each individual. This was expected since it used the most number of degrees of freedom. The category-fit reduced the number of degrees of freedom by fitting the data to two categories (sex and age). As a result, the residuals were larger than the unpooled fit, but still smaller than the population-fit, which estimated just one set of parameters for all individuals. The category-fit might be a good compromise between the unpooled and pooled fitting provided that any hierarchical model exists within your data.

### Estimate a Parameter from the Yeast G Protein Model

This example uses the yeast heterotrimeric G protein model and experimental data reported by [1]. For details about the model, see the **Background** section in “Parameter Scanning, Parameter Estimation, and Sensitivity Analysis in the Yeast Heterotrimeric G Protein Cycle”.

Load the G protein model.

```
sbioloadproject gprotein
```

Store the experimental data containing the time course for the fraction of active G protein.

```
time = [0 10 30 60 110 210 300 450 600]';  
GaFracExpt = [0 0.35 0.4 0.36 0.39 0.33 0.24 0.17 0.2]';
```

Create a `groupedData` object based on the experimental data.

```
tbl = table(time,GaFracExpt);  
grpData = groupedData(tbl);
```

Map the appropriate model component to the experimental data. In other words, indicate which species in the model corresponds to which response variable in the data. In this example, map the model parameter `GaFrac` to the experimental data variable `GaFracExpt` from `grpData`.

```
responseMap = 'GaFrac = GaFracExpt';
```

Use an `estimatedInfo` object to define the model parameter `kGd` as a parameter to be estimated.

```
estimatedParam = estimatedInfo('kGd');
```

Perform the parameter estimation.

```
fitResult = sbiofit(m1,grpData,responseMap,estimatedParam);
```

View the estimated parameter value of `kGd`.

```
fitResult.ParameterEstimates
```

```
ans=1×3 table  
      Name      Estimate      StandardError  
-----  
'kGd'      0.11307      3.4439e-05
```

Suppose you want to plot the model simulation results using the estimated parameter value. You can either rerun the `sbiofit` function and specify to return the optional

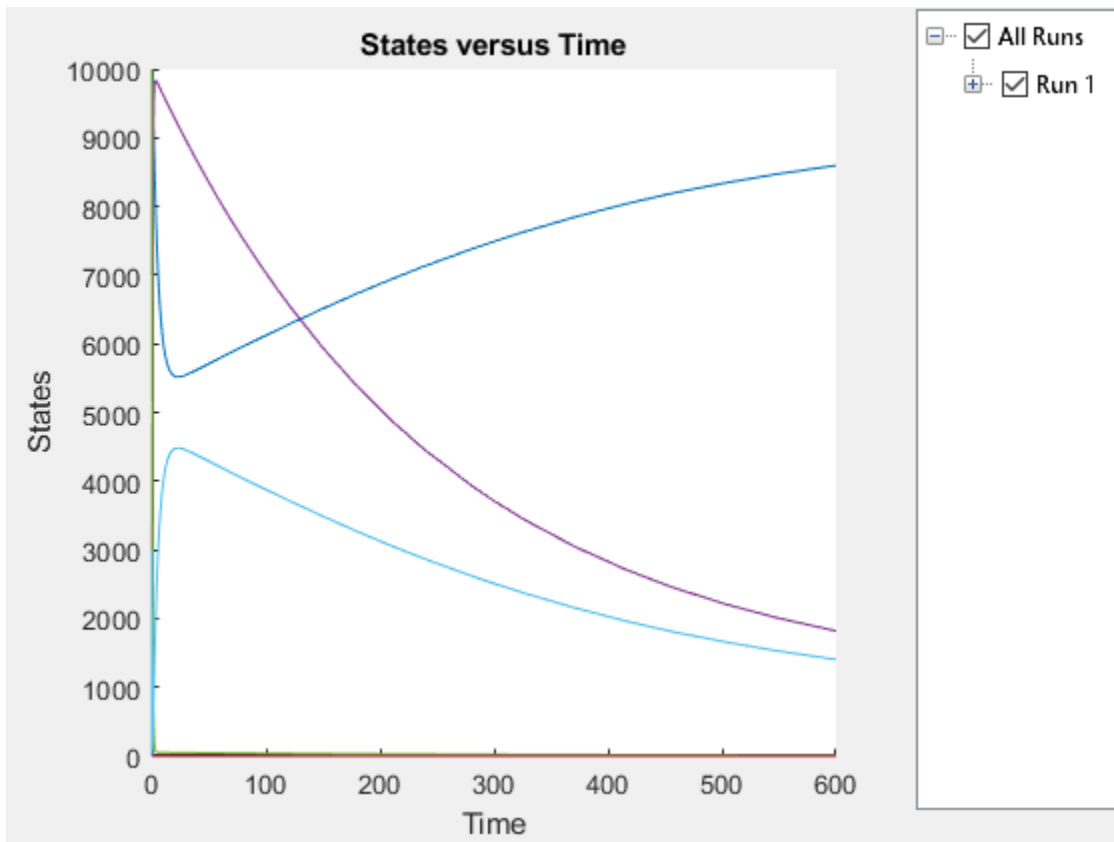


second output argument, which contains simulation results, or use the `fitted` method to retrieve the results without rerunning `sbiofit`.

```
[yfit,paramEstim] = fitted(fitResult);
```

Plot the simulation results.

```
sbioplot(yfit);
```



### Estimate Time Lag and Duration of a Dose

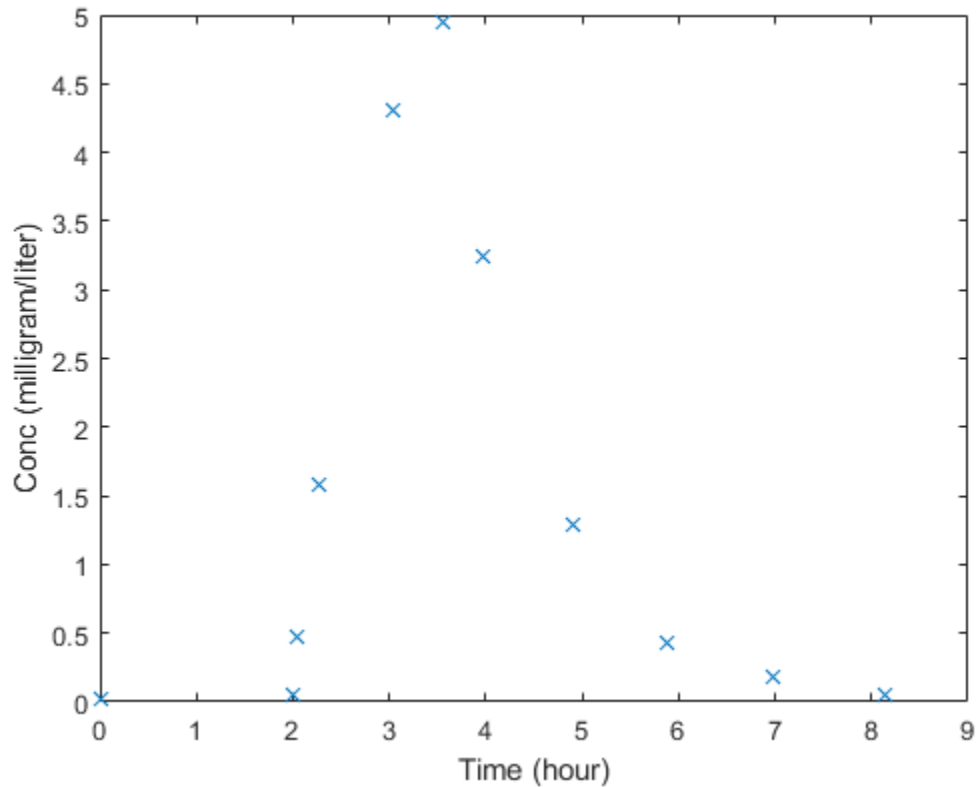
This example shows how to estimate the time lag before a bolus dose was administered and the duration of the dose using a one-compartment model.

Load a sample data set.

```
load lagDurationData.mat
```

Plot the data.

```
plot(data.Time,data.Conc,'x')  
xlabel('Time (hour)')  
ylabel('Conc (milligram/liter)')
```



Convert to groupedData.

```
gData = groupedData(data);  
gData.Properties.VariableUnits = {'hour','milligram/liter'};
```

Create a one-compartment model.

```
pkmd          = PKModelDesign;
pkc1          = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
model         = construct(pkmd);
configset     = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

Add two parameters that represent the time lag and duration of a dose. The lag parameter specifies the time lag before the dose is administered. The duration parameter specifies the length of time it takes to administer a dose.

```
lagP = addparameter(model, 'lagP');
lagP.ValueUnits = 'hour';
durP = addparameter(model, 'durP');
durP.ValueUnits = 'hour';
```

Create a dose object. Set the LagParameterName and DurationParameterName properties of the dose to the names of the lag and duration parameters, respectively.

```
dose          = sbiodose('dose');
dose.TargetName = 'Drug_Central';
dose.StartTime = 0;
dose.Amount    = 10;
dose.AmountUnits = 'milligram';
dose.TimeUnits = 'hour';
dose.LagParameterName = 'lagP';
dose.DurationParameterName = 'durP';
```

Map the model species to the corresponding data.

```
responseMap = {'Drug_Central = Conc'};
```

Specify the lag and duration parameters as parameters to estimate. Log-transform the parameters. Initialize them to 2 and set the upper bound and lower bound.

```
paramsToEstimate = {'log(lagP)', 'log(durP)'};
estimatedParams  = estimatedInfo(paramsToEstimate, 'InitialValue', 2, 'Bounds', [1 5]);
```

Perform parameter estimation.

```
fitResults = sbiofit(model, gData, responseMap, estimatedParams, dose, 'fminsearch')
```

```
fitResults =  
  OptimResults with properties:  
  
      ExitFlag: 1  
      Output: [1x1 struct]  
      GroupName: One group  
      Beta: [2x4 table]  
      ParameterEstimates: [2x4 table]  
      J: [11x2 double]  
      COVB: [2x2 double]  
      CovarianceMatrix: [2x2 double]  
      R: [11x1 double]  
      MSE: 0.0024  
      SSE: 0.0213  
      Weights: []  
      LogLikelihood: 18.7511  
      AIC: -33.5023  
      BIC: -32.7065  
      DFE: 9  
      DependentFiles: {1x2 cell}  
      EstimatedParameterNames: {'lagP' 'durP'}  
      ErrorModelInfo: [1x3 table]  
      EstimationFunction: 'fminsearch'
```

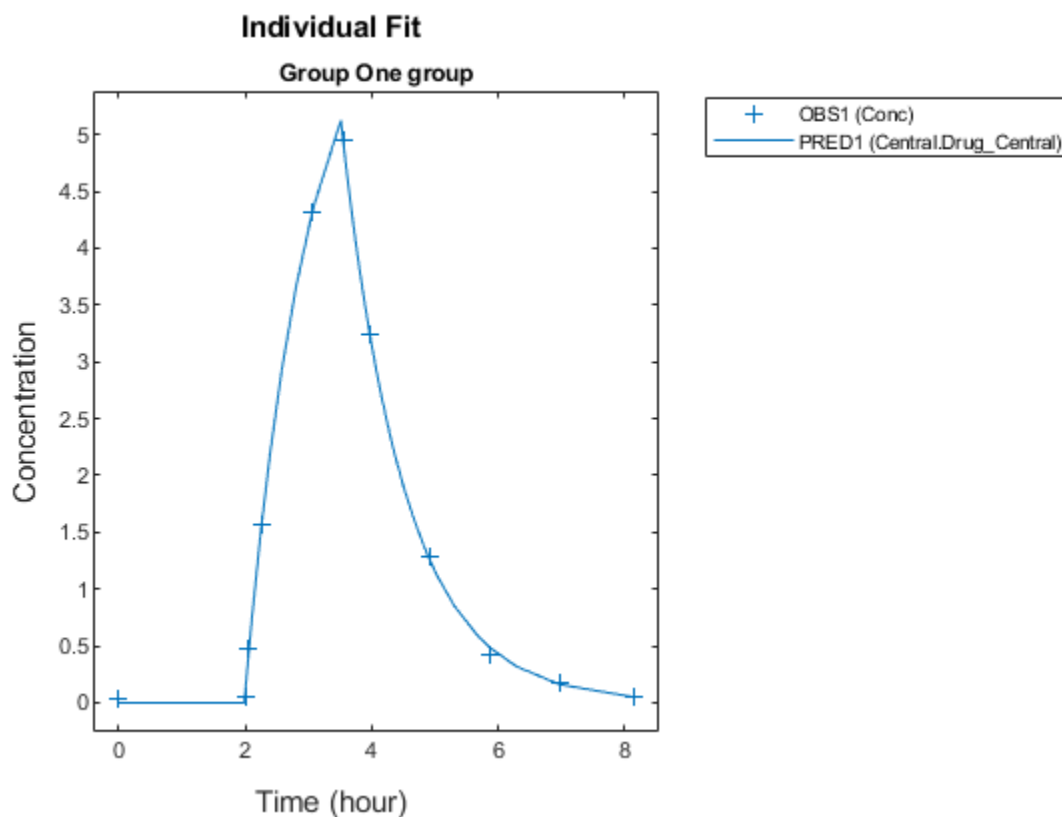
Display the result.

```
fitResults.ParameterEstimates
```

```
ans=2x4 table
```

Name	Estimate	StandardError	Bounds	
'lagP'	1.986	0.0051568	1	5
'durP'	1.527	0.012956	1	5

```
plot(fitResults)
```



## Input Arguments

### **sm** — SimBiology model

SimBiology model object

SimBiology model, specified as a SimBiology model object. The active configset object of the model contains solver settings for simulation. Any active doses and variants are applied to the model during simulation unless specified otherwise using the dosing and variants input arguments, respectively.

## **grpData — Data to fit**

groupedData object

Data to fit, specified as a groupedData object.

The name of the time variable must be defined in the IndependentVariableName property of grpData. For instance, if the time variable name is 'TIME', then specify it as follows.

```
grpData.Properties.IndependentVariableName = 'TIME';
```

If the data contains more than one group of measurements, the grouping variable name must be defined in the GroupVariableName property of grpData. For example, if the grouping variable name is 'GROUP', then specify it as follows.

```
grpData.Properties.GroupVariableName = 'GROUP';
```

A group usually refers to a set of measurements that represent a single time course, often corresponding to a particular individual, or experimental condition.

---

**Note** sbiofit uses the categorical function to identify groups. If any group values are converted to the same value by categorical, then those observations are treated as belonging to the same group. For instance, if some observations have no group information (that is, an empty character vector ''), then categorical converts empty character vectors to <undefined>, and these observations are treated as one group.

---

## **responseMap — Mapping information of model components to grpData**

character vector | string | string vector | cell array of character vectors

Mapping information of model components to grpData, specified as a character vector, string, string vector, or cell array of character vectors.

Each character vector or string is an equation-like expression, similar to assignment rules in SimBiology. It contains the name (or qualified name) of a quantity (species, compartment, or parameter) in the model sm, followed by the character '=' and the name of a variable in grpData. For clarity, white spaces are allowed between names and '='.

For example, if you have the concentration data 'CONC' in grpData for a species 'Drug\_Central', you can specify it as follows.

```
responseMap = 'Drug_Central = CONC';
```

To name a species unambiguously, use the qualified name, which includes the name of the compartment. To name a reaction-scoped parameter, use the reaction name to qualify the parameter. If the name is not a valid MATLAB variable name, surround it by square brackets such as `[reaction 1].[parameter 1]`.

An error is issued if any (qualified) name matches two components of the same type. However, you can use a (qualified) name that matches two components of different types, and the function first finds the species with the given name, followed by compartments and then parameters.

### **estiminfo — Estimated parameters**

`estimatedInfo` object | vector of `estimatedInfo` objects

Estimated parameters, specified as an `estimatedInfo` object or vector of `estimatedInfo` objects that defines the estimated parameters in the model `sm`, and other optional information such as their initial estimates, transformations, bound constraints, and categories. Supported transforms are `log`, `logit`, and `probit`.

You can specify bounds for all estimation methods. The lower bound must be less than the upper bound. For details, see “boundValues” on page 2-0 .

When using `scattersearch`, you must specify finite transformed bounds for each estimated parameter.

When using `fminsearch`, `nlinfit`, or `fminunc` with bounds, the objective function returns `Inf` if bounds are exceeded. When you turn on options such as `FunValCheck`, the optimization might error if bounds are exceeded during estimation. If using `nlinfit`, it might report warnings about the Jacobian being ill-conditioned or not being able to estimate if the final result is too close to the bounds.

If you do not specify `Pooled` name-value pair argument, `sbiofit` uses `CategoryVariableName` property of `estiminfo` to decide if parameters must be estimated for each individual, group, category, or all individuals as a whole. Use the `Pooled` option to override any `CategoryVariableName` values. For details about `CategoryVariableName` property, see `estimatedInfo` object.

---

**Note** `sbiofit` uses the `categorical` function to identify groups or categories. If any group values are converted to the same value by `categorical`, then those observations are treated as belonging to the same group. For instance, if some observations have no

group information (that is, an empty character vector '' as a group value), then `categorical` converts empty character vectors to `<undefined>`, and these observations are treated as one group.

---

## **dosing** — Dosing information

[ ] | 2-D matrix of dose objects

Dosing information, specified as an empty array, or 2-D matrix of dose objects (`ScheduleDose` object or `RepeatDose` object).

- If empty, no doses are applied during simulation, even if the model has active doses.
- If not empty, the matrix must have a single row or one row per group in the input data. If it has a single row, same doses are applied to all groups during simulation. If it has multiple rows, each row is applied to a separate group, in the same order as the groups appear in the input data.
- Multiple columns are allowed so that you can apply multiple dose objects to each group. All doses in a column must reference the same components in the model. For example, doses in the same column must have the same dose target (`TargetName`). If you parameterize any property of a dose, then all doses within the column must have this property parameterized to the same parameter. If some groups require more doses than others, then fill in the matrix with default (dummy) doses.
- A default dose has default values for all properties, except for the `Name` property. Create a default dose as follows.

```
d1 = sbiodose('d1');
```

- In addition to manually constructing dose objects using `sbiodose`, if `grpData` has dosing information, you can use the `createDoses` method to construct doses.

## **functionName** — Estimation function name

character vector | string

Estimation function name, specified as a character vector or string. Choices are as follows.

- 'fminsearch'
- 'nlinfit' (Statistics and Machine Learning Toolbox is required.)
- 'fminunc' (Optimization Toolbox is required.)
- 'fmincon' (Optimization Toolbox is required.)



- 'lsqcurvefit' (Optimization Toolbox is required.)
- 'lsqnonlin' (Optimization Toolbox is required.)
- 'patternsearch' (Global Optimization Toolbox is required.)
- 'ga' (Global Optimization Toolbox is required.)
- 'particleswarm' (Global Optimization Toolbox is required.)
- 'scattersearch' on page 1-109

By default, `sbiofit` uses the first available estimation function among the following: `lsqnonlin`, `nlinfit`, or `fminsearch`. The same priority applies to the default local solver choice for `scattersearch`.

For the summary of supported methods and fitting options, see “Supported Methods for Parameter Estimation”.

### **options — Options specific to estimation function**

struct | `optimoptions` object

Options specific to the estimation function, specified as a struct or `optimoptions` object.

- `statset` struct for `nlinfit`
- `optimset` struct for `fminsearch`
- `optimoptions` object for `lsqcurvefit`, `lsqnonlin`, `fmincon`, `fminunc`, `particleswarm`, `ga`, and `patternsearch`
- struct for `scattersearch`

See “Default Options for Estimation Algorithms” on page 1-107 for more details and default options associated with each estimation function.

### **variants — Variants**

[] | vector of variant objects

Variants, specified as an empty array or vector of variant objects. If empty, no variants are used even if the model has active variants.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'ErrorModel', 'constant', 'UseParallel', true` specifies a constant error model and to run simulations in parallel during parameter estimation.

### **ErrorModel — Error model**

`'constant'` (default) | character vector | string | string vector | cell array of character vector | categorical vector | table

Error models used for estimation, specified as a character vector, string, string vector, cell array of character vectors, categorical vector, or table.

If it is a table, it must contain a single variable that is a column vector of error model names. The names can be a cell array of character vectors, string vector, or a vector of categorical variables. If the table has more than one row, then the `RowNames` property must match the response variable names specified in the right hand side of `responseMap`. If the table does not use the `RowNames` property, the  $n$ th error is associated with the  $n$ th response.

If you specify only one error model, then `sbiofit` estimates one set of error parameters for all responses.

If you specify multiple error models using a categorical vector, string vector, or cell array of character vectors, the length of the vector or cell array must match the number of responses in the `responseMap` argument.

You can specify multiple error models only if you are using these methods: `lsqnonlin`, `lsqcurvefit`, `fmincon`, `fminunc`, `fminsearch`, `patternsearch`, `ga`, and `particleswarm`.

There are four built-in error models. Each model defines the error using a standard mean-zero and unit-variance (Gaussian) variable  $e$ , simulation results  $f$ , and one or two parameters  $a$  and  $b$ .

- `'constant'`:  $y = f + ae$
- `'proportional'`:  $y = f + b|f|e$
- `'combined'`:  $y = f + (a + b|f|)e$

- 'exponential':  $y = f * \exp(ae)$

---

**Note** If you specify an error model, you cannot specify weights except for the constant error model.

---

### Weights — Weights

matrix | function handle

Weights used for estimation, specified as a matrix of real positive weights, where the number of columns corresponds to the number of responses, or a function handle that accepts a vector of predicted response values and returns a vector of real positive weights.

If you specify an error model, you cannot use weights except for the constant error model. If neither the 'ErrorModel' or 'Weights' is specified, by default the function uses the constant error model with equal weights.

### Pooled — Fit option flag

false (default) | true

Fit option flag, specifying whether to fit each individual (`false`) or pool all individual data (`true`).

When `true`, the function performs fitting for all individuals or groups simultaneously using the same parameter estimates, and `fitResults` is a scalar results object.

When `false`, the function fits each group or individual separately using group- or individual-specific parameters, and `fitResults` is a vector of results objects with one result for each group.

---

**Note** Use this option to override any `CategoryVariableName` values of `estiminfo`.

---

### UseParallel — Flag for parallel simulations

false (default) | true

Flag to enable parallelization, specified as `true` or `false`. If `true` and Parallel Computing Toolbox™ is available, `sbiofit` supports several levels of parallelization, but only one level is used at a time.

For an unpooled fit (`'Pooled' = false`) for multiple groups, each fit is run in parallel.

For a pooled fit (`'Pooled' = true`), parallelization happens at the solver level. In other words, solver computations, such as objective function evaluations, are run in parallel. For details, see “Multiple Parameter Estimations in Parallel” on page 1-115.

### **SensitivityAnalysis — Flag to use parameter sensitivities to determine gradients of the objective function**

`true | false`

Flag to use parameter sensitivities to determine gradients of the objective function, specified as `true` or `false`. By default, it is `true` for `fmincon`, `fminunc`, `lsqnonlin`, and `lsqcurvefit` methods. Otherwise, it is `false`. If it is `true`, SimBiology always uses the `sundials` solver, regardless of what you have selected as the `SolverType` property in the `Configset` object.

SimBiology uses the complex-step approximation method to calculate parameter sensitivities. Such calculated sensitivities can be used to determine gradients of the objective function during parameter estimation to improve fitting. The default behavior of `sbiofit` is to use such sensitivities to determine gradients whenever the algorithm is gradient-based and if the SimBiology model supports sensitivity analysis. For details about the model requirements and sensitivity analysis, see “Sensitivity Calculation”.

### **ProgressPlot — Flag to show the progress of parameter estimation**

`false (default) | true`

Flag to show the progress of parameter estimation, specified as `true` or `false`. If `true`, a new figure opens containing plots that show the log-likelihood, termination criteria, and estimated parameters for each iteration. This option is not supported for the `nlinfit` method.

If you are using the Optimization Toolbox functions (`fminunc`, `fmincon`, `lsqcurvefit`, `lsqnonlin`), the figure also shows the First Order Optimality (Optimization Toolbox) plot.

For an unpooled fit, each line on the plots represents an individual. For a pooled fit, a single line represents all individuals. The line becomes faded when the fit is complete. The plots also keep track of the progress when you run `sbiofit` (for both pooled and unpooled fits) in parallel using remote clusters. For details, see “Progress Plot”.

## Output Arguments

### **fitResults** — Estimation results

`OptimResults` object | `NLINResults` object | Vector of results objects

Estimation results, returned as a `OptimResults` object or `NLINResults` object or a vector of these objects. Both results objects are subclasses of the `LeastSquaresResults` object.

If the function uses `nlinfit`, then `fitResults` is a `NLINResults` object. Otherwise, `fitResults` is an `OptimResults` object.

When 'Pooled' is set to `false`, the function fits each group separately using group-specific parameters, and `fitResults` is a vector of results objects with one results object for each group.

When 'Pooled' is set to `true`, the function performs fitting for all individuals or groups simultaneously using the same parameter estimates, and `fitResults` is a scalar results object.

When 'Pooled' is not used, and `CategoryVariableName` values of `estiminfo` are all `<none>`, `fitResults` is a single result object. This is the same behavior as setting 'Pooled' to `true`.

When 'Pooled' is not used, and `CategoryVariableName` values of `estiminfo` are all `<GroupVariableName>`, `fitResults` is a vector of results objects. This is the same behavior as setting 'Pooled' to `false`.

In all other cases, `fitResults` is a scalar object containing estimated parameter values for different groups or categories specified by `CategoryVariableName`.

### **simdata** — Simulation results

Vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects representing simulation results for each group or individual.

If the 'Pooled' option is `false`, then each simulation uses group-specific parameter values. If `true`, then all simulations use the same (population-wide) parameter values.

The states reported in `simdata` are the states that were included in the `responseMap` input argument and any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the SimBiology model `sm`.

## Definitions

### Maximum Likelihood Estimation

SimBiology estimates parameters by the method of maximum likelihood. Rather than directly maximizing the likelihood function, SimBiology constructs an equivalent minimization problem. Whenever possible, the estimation is formulated as a weighted least squares (**WLS**) optimization that minimizes the sum of the squares of weighted residuals. Otherwise, the estimation is formulated as the minimization of the negative of the logarithm of the likelihood (**NLL**). The **WLS** formulation often converges better than the **NLL** formulation, and SimBiology can take advantage of specialized **WLS** algorithms, such as the Levenberg-Marquardt algorithm implemented in `lsqnonlin` and `lsqcurvefit`. SimBiology uses **WLS** when there is a single error model that is constant, proportional, or exponential. SimBiology uses **NLL** if you have a combined error model or a multiple-error model, that is, a model having an error model for each response.

`sbiofit` supports different optimization methods, and passes in the formulated **WLS** or **NLL** expression to the optimization method that minimizes it.

	Expression that is being minimized
Weighted Least Squares ( <b>WLS</b> )	For the constant error model, $\sum_i^N (y_i - f_i)^2$
	For the proportional error model, $\sum_i^N \frac{(y_i - f_i)^2}{f_i^2 / f_{gm}^2}$
	For the exponential error model, $\sum_i^N (\ln y_i - \ln f_i)^2$

	Expression that is being minimized
	For numeric weights, $\sum_j \frac{(y_i - f_i)^2}{w_{gm}/w_i}$
Negative Log-likelihood (NLL)	For the combined error model and multiple-error model, $\sum_i \frac{(y_i - f_i)^2}{2\sigma_i^2} + \sum_i \ln \sqrt{2\pi\sigma_i^2}$

The variables are defined as follows.

$N$	Number of experimental observations
$y_i$	The $i$ th experimental observation
$f_i$	Predicted value of the $i$ th observation
$\sigma_i$	Standard deviation of the $i$ th observation. <ul style="list-style-type: none"> <li>• For the constant error model, <math>\sigma_i = a</math></li> <li>• For the proportional error model, <math>\sigma_i = b f_i </math></li> <li>• For the combined error model, <math>\sigma_i = a + b f_i </math></li> </ul>
$f_{gm}$	$f_{gm} = \left( \prod_i  f_i  \right)^{1/N}$
$w_i$	The weight of the $i$ th predicted value
$w_{gm}$	$w_{gm} = \left( \prod_i w_i \right)^{1/N}$

When you use numeric weights or the weight function, the weights are assumed to be

inversely proportional to the variance of the error, that is,  $\sigma_i^2 = \frac{a^2}{w_i}$  where  $a$  is the constant error parameter. If you use weights, you cannot specify an error model except the constant error model.

Various optimization methods have different requirements on the function that is being minimized. For some methods, the estimation of model parameters is performed independently of the estimation of the error model parameters. The following table summarizes the error models and any separate formulas used for the estimation of error model parameters, where  $a$  and  $b$  are error model parameters and  $e$  is the standard mean-zero and unit-variance (Gaussian) variable.

Error Model	Error Parameter Estimation Function
'constant': $y_i = f_i + ae$	$a^2 = \frac{1}{N} \sum_i (y_i - f_i)^2$
'exponential': $y_i = f_i \exp(ae)$	$a^2 = \frac{1}{N} \sum_i (\ln y_i - \ln f_i)^2$
'proportional': $y_i = f_i + b f_i e$	$b^2 = \frac{1}{N} \sum_i \left( \frac{y_i - f_i}{f_i} \right)^2$
'combined': $y_i = f_i + (a + b f_i )e$	Error parameters are included in the minimization.
Weights	$a^2 = \frac{1}{N} \sum_i (y_i - f_i)^2 w_i$

---

**Note** `nlinfit` only support single error models, not multiple-error models, that is, response-specific error models. For a combined error model, it uses an iterative **WLS**



algorithm. For other error models, it uses the **WLS** algorithm as described previously. For details, see `nlinfit`.

## Default Options for Estimation Algorithms

Function	Default Options
<code>nlinfit</code>	<p><code>sbiofit</code> uses the default options structure associated with <code>nlinfit</code>, except for:</p> <p><code>FunValCheck</code> = 'off'</p> <p><code>DerivStep</code> = <math>\max(\text{eps}^{(1/3)}, \min(1e-4, \text{SolverOptions.RelativeTolerance}))</math>, where <code>SolverOptions</code> property corresponds to the model's active <code>configset</code> object.</p>
<code>fmincon</code>	<p><code>sbiofit</code> uses the default options structure associated with <code>fmincon</code>, except for:</p> <p><code>Display</code> = 'off'</p> <p><code>FunctionTolerance</code> = <math>1e-6 * f_0</math>, where <code>f0</code> is the initial value of objective function.</p> <p><code>OptimalityTolerance</code> = <math>1e-6 * f_0</math>, where <code>f0</code> is the initial value of objective function.</p> <p><code>Algorithm</code> = 'trust-region-reflective' when 'SensitivityAnalysis' is true, or 'interior-point' when 'SensitivityAnalysis' is false.</p> <p><code>FiniteDifferenceStepSize</code> = <math>\max(\text{eps}^{(1/3)}, \min(1e-4, \text{SolverOptions.RelativeTolerance}))</math>, where <code>SolverOptions</code> property corresponds to the model active <code>configset</code> object.</p> <p><code>TypicalX</code> = <math>1e-6 * x_0</math>, where <code>x0</code> is an array of transformed initial estimates.</p>

Function	Default Options
fminunc	<p>sbiofit uses the default options structure associated with fminunc, except for:</p> <p>Display = 'off'</p> <p>FunctionTolerance = <math>1e-6 \cdot f_0</math>, where <math>f_0</math> is the initial value of objective function.</p> <p>OptimalityTolerance = <math>1e-6 \cdot f_0</math>, where <math>f_0</math> is the initial value of objective function.</p> <p>Algorithm = 'trust-region' when 'SensitivityAnalysis' is true, or 'quasi-newton' when 'SensitivityAnalysis' is false.</p> <p>FiniteDifferenceStepSize = <math>\max(\text{eps}^{(1/3)}, \min(1e-4, \text{SolverOptions.RelativeTolerance}))</math>, where SolverOptions property corresponds to the model active configset object.</p> <p>TypicalX = <math>1e-6 \cdot x_0</math>, where <math>x_0</math> is an array of transformed initial estimates.</p>
fminsearch	<p>sbiofit uses the default options structure associated with fminsearch, except for:</p> <p>Display = 'off'</p> <p>TolFun = <math>1e-6 \cdot f_0</math>, where <math>f_0</math> is the initial value of objective function.</p>
lsqcurvefit, lsqnonlin	<p>Requires Optimization Toolbox.</p> <p>sbiofit uses the default options structure associated with lsqcurvefit and lsqnonlin, except for:</p> <p>Display = 'off'</p> <p>FunctionTolerance = <math>1e-6 \cdot \text{norm}(f_0)</math>, where <math>f_0</math> is the initial value of objective function.</p> <p>OptimalityTolerance = <math>1e-6 \cdot f_0</math>, where <math>f_0</math> is the initial value of objective function.</p> <p>FiniteDifferenceStepSize = <math>\max(\text{eps}^{(1/3)}, \min(1e-4, \text{SolverOptions.RelativeTolerance}))</math>, where SolverOptions property corresponds to the model active configset object.</p> <p>TypicalX = <math>1e-6 \cdot x_0</math>, where <math>x_0</math> is an array of transformed initial estimates.</p>

Function	Default Options
patternsearch	Requires Global Optimization Toolbox.  sbiofit uses the default options object (optimoptions) associated with patternsearch, except for: Display = 'off' FunctionTolerance = $1e-6 * f_0$ , where $f_0$ is the initial value of the objective function. MeshTolerance = $1.0e-3$ AccelerateMesh = true
ga	Requires Global Optimization Toolbox.  sbiofit uses the default options object (optimoptions) associated with ga, except for: Display = 'off' FunctionTolerance = $1e-6 * f_0$ , where $f_0$ is the initial value of objective function. MutationFcn = @mutationadaptfeasible
particleswarm	Requires Global Optimization Toolbox. sbiofit uses the following default options for the particleswarm algorithm, except for: Display = 'off' FunctionTolerance = $1e-6 * f_0$ , where $f_0$ is the initial value of objective function. InitialSwarmSpan = 2000 or 8; 2000 for estimated parameters with no transform; 8 for estimated parameters with log, logit, or probit transforms.
scattersearch	See "Scatter Search Algorithm" on page 1-109.

## Scatter Search Algorithm

The scattersearch method implements a global optimization algorithm [2] that addresses some challenges of parameter estimation in dynamic models, such as convergence to local minima.

The algorithm selects a subset of points from an initial pool of points. In that subset, some points are the best in quality (that is, lowest function value) and some are randomly selected. The algorithm iteratively evaluates the points and explores different directions

around various solutions to find better solutions. During this iteration step, the algorithm replaces any old solution with a new one of better quality. Iterations proceed until any stopping criteria are met. It then runs a local solver on the best point.

### Initialization

To start the scatter search, the algorithm first decides the total number of points needed (`NumInitialPoints`). By default, the total is  $10 \cdot N$ , where  $N$  is the number of estimated parameters. It selects `NumInitialPoints` points (rows) from `InitialPointMatrix`. If `InitialPointMatrix` does not have enough points, the algorithm calls the function defined in `CreationFcn` to generate the additional points needed. By default, Latin hypercube sampling is used to generate these additional points. The algorithm then selects a subset of `NumTrialPoints` points from `NumInitialPoints` points. A fraction (`FractionInitialBest`) of the subset contains the best points in terms of quality. The remaining points in the subset are randomly selected.

### Iteration Steps

The algorithm iterates on the points in the subset as follows:

- 1 Define hyper-rectangles around each pair of points by using the relative qualities (that is, function values) of these points as a measure of bias to create these rectangles.
- 2 Evaluate a new solution inside each rectangle. If the new solution outperforms the original solution, replace the original with the new one.
- 3 Apply the go-beyond strategy to the improved solutions and exploit promising directions to find better solutions.
- 4 Run a local search at every `LocalSearchInterval` iteration. Use the `LocalSelectBestProbability` probability to select the best point as the starting point for a local search. By default, the decision is random, giving an equal chance to select the best point or a random point from the trial points. If the new solution outperforms the old solution, replace the old one with the new one.
- 5 Replace any stalled point that does not produce any new outperforming solution after `MaxStallTime` seconds with another point from the initial set.
- 6 Evaluate stopping criteria. Stop iterating if any criteria are met.

The algorithm then runs a local solver on the best point seen.

### Stopping Criteria

The algorithm iterates until it reaches a stopping criterion.

Stopping Option	Stopping Test
FunctionTolerance and MaxStallIterations	Relative change in best objective function value over the last MaxStallIterations is less than FunctionTolerance.
MaxIterations	Number of iterations reaches MaxIterations.
OutputFcn	OutputFcn can halt the iterations.
ObjectiveLimit	Best objective function value at an iteration is less than or equal to ObjectiveLimit.
MaxStallTime	Best objective function value did not change in the last MaxStallTime seconds.
MaxTime	Function run time exceeds MaxTime seconds.

You create the options for the algorithm using a struct.

Option	Description
CreationFcn	Handle to the function that creates additional points needed for the algorithm. Default is the character vector 'auto', which uses Latin hypercube sampling.  The function signature is: <code>points = CreationFcn(s,N,lb,ub)</code> , where <code>s</code> is the total number of sampled points, <code>N</code> is the number of estimated parameters, <code>lb</code> is the lower bound, and <code>ub</code> is the upper bound. If any output from the function exceeds bounds, these results are truncated to the bounds.
Display	Level of display returned to the command line. <ul style="list-style-type: none"> <li>• 'off' or 'none' (default) displays no output.</li> <li>• 'iter' gives iterative display.</li> <li>• 'final' displays just the final output.</li> </ul>
FractionInitialBest	Numeric scalar from 0 through 1. Default is 0.5. This number is the fraction of the NumTrialPoints that are selected as the best points from the NumInitialPoints points.

Option	Description
FunctionTolerance	Numeric scalar from 0 through 1. Default is 1e-6. The solver stops if the relative change in best objective function value over the last <code>MaxStallIterations</code> is less than <code>FunctionTolerance</code> . This option is also used to remove duplicate local solutions. See <code>XTolerance</code> for details.
InitialPointMatrix	<p>Initial (or partial) set of points. <math>M</math>-by-<math>N</math> real finite matrix, where <math>M</math> is the number of points and <math>N</math> is the number of estimated parameters.</p> <p>If <math>M &lt; \text{NumInitialPoints}</math>, then <code>scattersearch</code> creates more points so that the total number of rows is <code>NumInitialPoints</code>.</p> <p>If <math>M &gt; \text{NumInitialPoints}</math>, then <code>scattersearch</code> uses the first <code>NumInitialPoints</code> rows.</p> <p>Default is the initial transformed values of estimated parameters stored in the <code>InitialTransformedValue</code> property of the <code>estimatedInfo</code> object, that is, <code>[estiminfo.InitialTransformedValue]</code>.</p>
LocalOptions	<p>Options for the local solver. It can be a <code>struct</code> (created with <code>optimset</code> or <code>statset</code>) or an <code>optimoptions</code> object, depending on the local solver. Default is the character vector 'auto', which uses the default options of the selected solver with some exceptions on page 1-107. In addition to these exceptions, the following options limit the time spent in the local solver because it is called repeatedly:</p> <ul style="list-style-type: none"> <li>• <code>MaxFunEvals</code> (maximum number of function evaluations allowed) = 300</li> <li>• <code>MaxIter</code> (maximum number of iterations allowed) = 200</li> </ul>
LocalSearchInterval	Positive integer. Default is 10. The <code>scattersearch</code> algorithm applies the local solver to one of the trial points after the first iteration and again every <code>LocalSearchInterval</code> iteration.
LocalSelectBestProbability	Numeric scalar from 0 through 1. Default is 0.5. It is the probability of selecting the best point as the starting point for a local search. In other cases, one of the trial points is selected at random.

Option	Description
LocalSolver	<p>Character vector or string specifying the name of a local solver. Supported methods are 'fminsearch', 'lsqnonlin', 'lsqcurvefit', 'fmincon', 'fminunc', 'nlinfit'.</p> <p>Default local solver is selected with the following priority:</p> <ul style="list-style-type: none"> <li>• If Optimization Toolbox is available, the solver is <code>lsqnonlin</code>.</li> <li>• If Statistics and Machine Learning Toolbox is available, the solver is <code>nlinfit</code>.</li> <li>• Otherwise, the solver is <code>fminsearch</code>.</li> </ul>
MaxIterations	Positive integer. Default is the character vector 'auto' representing $20*N$ , where $N$ is the number of estimated parameters.
MaxStallIterations	Positive integer. Default is 50. The solver stops if the relative change in the best objective function value over the last <code>MaxStallIterations</code> iterations is less than <code>FunctionTolerance</code> .
MaxStallTime	Positive scalar. Default is <code>Inf</code> . The solver stops if <code>MaxStallTime</code> seconds have passed since the last improvement in the best-seen objective function value. Here, the time is the wall clock time as opposed to processor cycles.
MaxTime	Positive scalar. Default is <code>Inf</code> . The solver stops if <code>MaxTime</code> seconds have passed since the beginning of the search. The time here means the wall clock time as opposed to processor cycles.
NumInitialPoints	Positive integer that is $\geq$ <code>NumTrialPoints</code> . The solver generates <code>NumInitialPoints</code> points before selecting a subset of trial points ( <code>NumTrialPoints</code> ) for subsequent steps. Default is the character vector 'auto', which represents $10*N$ , where $N$ is the number of estimated parameters.
NumTrialPoints	Positive integer that is $\geq 2$ and $\leq$ <code>NumInitialPoints</code> . The solver generates <code>NumInitialPoints</code> initial points before selecting a subset of trial points ( <code>NumTrialPoints</code> ) for subsequent steps. Default is the character vector 'auto', which represents the first even number $n$ for which $n^2 - n \geq 10 * N$ , where $N$ is the number of estimated parameters.
ObjectiveLimit	Scalar. Default is <code>-Inf</code> . The solver stops if the best objective function value at an iteration is less than or equal to <code>ObjectiveLimit</code> .

Option	Description
OutputFunction	<p>Function handle or cell array of function handles. Output functions can read iterative data and stop the solver. Default is [].</p> <p>Output function signature is <code>stop = myfun(optimValues, state)</code>, where:</p> <ul style="list-style-type: none"> <li>• <code>stop</code> is a logical scalar. Set to <code>true</code> to stop the solver.</li> <li>• <code>optimValues</code> is a structure containing information about the trial points with fields. <ul style="list-style-type: none"> <li>• <code>bestx</code> is the best solution point found, corresponding to the function value <code>bestfval</code>.</li> <li>• <code>bestfval</code> is the best (lowest) objective function value found.</li> <li>• <code>iteration</code> is the iteration number.</li> <li>• <code>medianfval</code> is the mean objective function value among all the current trial points.</li> <li>• <code>stalliterations</code> is the number of iterations since the last change in <code>bestfval</code>.</li> <li>• <code>trialx</code> is a matrix of the current trial points. Each row represents one point, and the number of rows is equal to <code>NumTrialPoints</code>.</li> <li>• <code>trialfvals</code> is a vector of objective function values for trial points. It is a matrix for <code>lsqcurvefit</code> and <code>lsqnonlin</code> methods.</li> </ul> </li> <li>• <code>state</code> is a character vector giving the status of the current iteration. <ul style="list-style-type: none"> <li>• <code>'init'</code> - The solver has not begun to iterate. Your output function can use this state to open files, or set up data structures or plots for subsequent iterations.</li> <li>• <code>'iter'</code> - The solver is proceeding with its iterations. Typically, this state is where your output function performs its work.</li> <li>• <code>'done'</code> - The solver reaches a stopping criterion. Your output function can use this state to clean up, such as closing any files it opened.</li> </ul> </li> </ul>
TrialStallLimit	<p>Positive integer, with default value of 22. If a particular trial point does not improve after <code>TrialStallLimit</code> iterations, it is replaced with another point.</p>



Option	Description
UseParallel	Logical flag to compute objective function in parallel. Default is false.
XTolerance	Numeric scalar from 0 through 1. Default is 1e-6. This option defines how close two points must be to consider them identical for creating the vector of local solutions. The solver calculates the distance between a pair of points with norm, the Euclidean distance. If two solutions are within XTolerance distance of each other and have objective function values within FunctionTolerance of each other, the solver considers them identical. If both conditions are not met, the solver reports the solutions as distinct.  To get a report of every potential local minimum, set XTolerance to 0. To get a report of fewer results, set XTolerance to a larger value.

## Multiple Parameter Estimations in Parallel

There are two ways to use parallel computing for parameter estimation.

To enable parallelization for `sbiofit`, set the name-value pair 'UseParallel' to true. The function supports several levels of parallelization, but only one level is used at a time. For an unpooled fit for multiple groups (or individuals), each fit runs in parallel. For a pooled fit, parallelization happens at the solver level if the solver supports it. That is, `sbiofit` sets the parallel option of the corresponding estimation method (solver) to true, and the objection function evaluations are performed in parallel. For instance, gradients are computed in parallel for gradient-based methods. If you are performing a pooled fit with multiple groups using a solver that does not have the parallel option, the simulations run in parallel for each group during optimization (maximum likelihood estimation on page 1-104).

You can also call `sbiofit` inside a `parfor` loop or use a `parfeval` inside a `for`-loop to perform multiple parameter estimations in parallel. It is recommended that you use `parfeval` because these parallel estimations run asynchronously. If one fit produces an error, it does not affect the other fits.

If you are trying to find a global minimum, you can use global solvers, such as `particleswarm` or `ga` (Global Optimization Toolbox is required). However, if you want to define the initial conditions and run the fits in parallel, see the following example that shows how to use both `parfor` and `parfeval`.

## Model and Data Setup

Load the G protein model.

```
sbioloadproject gprotein
```

Store the experimental data containing the time course for the fraction of active G protein [1].

```
time = [0 10 30 60 110 210 300 450 600]';  
GaFracExpt = [0 0.35 0.4 0.36 0.39 0.33 0.24 0.17 0.2]';
```

Create a `groupedData` object based on the experimental data.

```
tbl = table(time,GaFracExpt);  
grpData = groupedData(tbl);
```

Map the appropriate model element to the experimental data.

```
responseMap = 'GaFrac = GaFracExpt';
```

Specify the parameter to estimate.

```
paramToEstimate = {'kGd'};
```

Generate initial parameter values for `kGd`.

```
rng('default');  
iniVal = abs(normrnd(0.01,1,10,1));  
fitResultPar = [];
```

## Parallel Pool Setup

Start a parallel pool using the local profile.

```
poolObj = parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...  
connected to 12 workers.
```

Send the model to each worker. The following code ensures that the model is distributed only once to each worker.

```
modelOnWorkers = parallel.pool.Constant(m1);
```

## Using `parfeval` (Recommended)

First, define a function handle that uses the local function `sbiofitpar` for estimation. Make sure the function `sbiofitpar` is defined at the end of the script.

```
optimfun = @(x) sbiofitpar(modelOnWorkers.Value,grpData,responseMap,x);
```

Perform multiple parameter estimations in parallel via `parfeval` using different initial parameter values.

```
for i=1:length(iniVal)
    f(i) = parfeval(optimfun,1,iniVal(i));
end
fitResultPar = fetchOutputs(f);
```

Summarize the results for each run.

```
allParValues = vertcat(fitResultPar.ParameterEstimates);
allParValues.LogLikelihood = [fitResultPar.LogLikelihood]';
allParValues.RunNumber = (1:length(iniVal))';
allParValues.Name = categorical(allParValues.Name);
allParValues.InitialValue = iniVal;
% Rearrange the columns.
allParValues = allParValues(:,[5 1 6 2 3 4]);
% Sort rows by LogLikelihood.
sortrows(allParValues,'LogLikelihood')
```

*ans=10×6 table*

RunNumber	Name	InitialValue	Estimate	StandardError	LogLikelihood
9	kGd	3.5884	3.0219	0.13319	-1.2842
10	kGd	2.7794	2.779	0.029701	-1.2319
3	kGd	2.2488	2.2488	0.096013	-1.0786
2	kGd	1.8439	1.844	0.032672	-0.90104
6	kGd	1.2977	1.2977	0.011344	-0.48209
4	kGd	0.87217	0.65951	0.003583	0.9279
1	kGd	0.54767	0.54777	0.0022074	1.5315
7	kGd	0.42359	0.42363	0.0024555	2.6097
8	kGd	0.35262	0.35291	0.00065289	3.6098
5	kGd	0.32877	0.32877	0.00042474	4.0604

Define the local function `sbiofitpar` that performs parameter estimation using `sbiofit`.

```
function fitresult = sbiofitpar(model,grpData,responseMap,initialValue)
estimatedParam = estimatedInfo('kGd');
```

```
estimatedParam.InitialValue = initialValue;  
fitresult = sbiofit(model,grpData,responseMap,estimatedParam);  
end
```

### Using parfor

Alternatively, you can perform multiple parameter estimations in parallel via the `parfor` loop.

```
parfor i=1:length(iniVal)  
    estimatedParam = estimatedInfo(paramToEstimate,'InitialValue',iniVal(i));  
    fitResultTemp = sbiofit(modelOnWorkers.Value,grpData,responseMap,estimatedParam);  
    fitResultPar = [fitResultPar;fitResultTemp];  
end
```

Close the parallel pool.

```
delete(poolObj);
```

## Parameter Estimation with Hybrid Solvers

`sbiofit` supports global optimization methods, namely `ga` and `particleswarm` (Global Optimization Toolbox required). To improve optimization results, these methods lets you run a hybrid function after the global solver stops. The hybrid function uses the final point returned by the global solver as its initial point. Supported hybrid functions are:

- `fminunc`
- `fmincon`
- `patternsearch`
- `fminsearch`

Make sure that your hybrid function accepts your problem constraints. That is, if your parameters are bounded, use an appropriate function (such as `fmincon` or `patternsearch`) for a constrained optimization. If not bounded, use `fminunc`, `fminsearch`, or `patternsearch`. Otherwise, `sbiofit` throws an error.

For an illustrated example, see [Perform Hybrid Optimization Using sbiofit](#).

## References

- [1] Yi, T-M., Kitano, H., and Simon, M. (2003). A quantitative characterization of the yeast heterotrimeric G protein cycle. PNAS. 100, 10764-10769.
- [2] Gábor, A., and Banga, J.R. (2015). Robust and efficient parameter estimation in dynamic models of biological systems. BMC Systems Biology. 9:74.

## Extended Capabilities

### Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set 'UseParallel' to true.

For more information, see the 'UseParallel' name-value pair argument.

### See Also

LeastSquaresResults object | NLINResults object | OptimResults object | estimatedInfo object | fmincon | fminsearch | fminunc | ga | groupedData object | lsqcurvefit | lsqnonlin | nlinfit | particleswarm | patternsearch | sbiofitmixed

### Topics

"Parameter Scanning, Parameter Estimation, and Sensitivity Analysis in the Yeast Heterotrimeric G Protein Cycle"

"What is Nonlinear Regression?"

"Fitting Options in SimBiology"

"Maximum Likelihood Estimation"

"Fitting Workflow for sbiofit"

"Supported Methods for Parameter Estimation"

"Sensitivity Calculation"

"Progress Plot"

**Introduced in R2014a**

# sbiofitmixed

Fit nonlinear mixed-effects model (requires Statistics and Machine Learning Toolbox software)

## Syntax

```
fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo)
fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,
dosing)
fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,
dosing,functionName)
fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,
dosing,functionName,opt)
fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,
dosing,functionName,opt,variants)
fitResults = sbiofitmixed(_,'UseParallel',tf_parallel)
fitResults = sbiofitmixed(_,'ProgressPlot',tf_progress)

[fitResults,simDataI,simDataP] = sbiofitmixed(_)
```

## Description

`fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo)` performs nonlinear mixed-effects estimation using the SimBiology model `sm` and returns a `NLMEResults` object `fitResults`.

`grpData` is a `groupedData` object specifying the data to fit. `responseMap` defines the mapping between the model components and response data in `grpData`. `covEstiminfo` is a `CovariateModel` object or an array of `estimatedInfo` objects that defines the parameters to be estimated.

If the model contains active doses and variants, they are applied during the simulation.

`fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,dosing)` uses the dosing information specified by a matrix of SimBiology dose objects `dosing` instead of using the active doses of the model `sm` if there are any.

`fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,dosing,functionName)` uses the estimation function specified by `functionName` that must be either `'nlmefit'` or `'nlmefitsa'`.

`fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,dosing,functionName,opt)` uses the additional options specified by `opt` for the estimation function `functionName`.

`fitResults = sbiofitmixed(sm,grpData,responseMap,covEstiminfo,dosing,functionName,opt,variants)` applies variant objects specified as `variants` instead of using any active variants of the model.

`fitResults = sbiofitmixed(_, 'UseParallel', tf_parallel)` specifies whether to estimate parameters in parallel if Parallel Computing Toolbox is available.

`fitResults = sbiofitmixed(_, 'ProgressPlot', tf_progress)` specifies whether to show the progress of parameter estimation.

`[fitResults,simDataI,simDataP] = sbiofitmixed(_)` returns a vector of results objects `fitResults`, vector of simulation results `simDataI` using individual-specific parameter estimates, and vector of simulation results `simDataP` using population parameter estimates.

---

## Note

- `sbiofitmixed` unifies `sbionlmefit` and `sbionlmefitsa` estimation functions. Use `sbiofitmixed` to perform nonlinear mixed-effects modeling and estimation.
  - `sbiofitmixed` simulates the model using a `SimFunction` object, which automatically accelerates simulations by default. Hence it is not necessary to run `sbioaccelerate` before you call `sbiofitmixed`.
- 

## Examples

### Fit a One-Compartment PK Model to the Phenobarbital Data

This example uses data collected on 59 preterm infants given phenobarbital during the first 16 days after birth [1]. Each infant received an initial dose followed by one or more



sustaining doses by intravenous bolus administration. A total of between 1 and 6 concentration measurements were obtained from each infant at times other than dose times, for a total of 155 measurements. Infant weights and APGAR scores (a measure of newborn health) were also recorded.

Load the data.

```
load pheno.mat ds
```

Convert the dataset to a `groupedData` object, a container for holding tabular data that is divided into groups. It can automatically identify commonly used variable names as the grouping variable or independent (time) variable. Display the properties of the data and confirm that `GroupVariableName` and `IndependentVariableName` are correctly identified as `'ID'` and `'TIME'`, respectively.

```
data = groupedData(ds);
data.Properties

ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Observations' 'Variables'}
    VariableNames: {'ID' 'TIME' 'DOSE' 'WEIGHT' 'APGAR' 'CONC'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
    RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: 'ID'
    IndependentVariableName: 'TIME'
```

Create a simple one-compartment PK model with bolus dosing and linear clearance to fit such data. Use the `PKModelDesign` object to construct the model. Each compartment is defined by a name, dosing type, a clearance type, and whether or not the dosing requires a lag parameter. After constructing the model, you can also get a `PKModelMap` object `map` that lists the names of species and parameters in the model that are most relevant for fitting.

```
pkmd = PKModelDesign;
addCompartment(pkmd, 'Central', 'DosingType', 'Bolus', ...
    'EliminationType', 'linear-clearance', ...
    'HasResponseVariable', true, 'HasLag', false);
[onecomp, map] = pkmd.construct;
```

Describe the experimentally measured response by mapping the appropriate model component to the response variable. In other words, indicate which species in the model corresponds to which response variable in the data. The PKModelMap property `Observed` indicates that the relevant species in the model is `Drug_Central`, which represents the drug concentration in the system. The relevant data variable is `CONC`, which you visualized previously.

```
map.Observed
```

```
ans = 1x1 cell array  
    {'Drug_Central'}
```

Map the `Drug_Central` species to the `CONC` variable.

```
responseMap = 'Drug_Central = CONC';
```

The parameters to estimate in this model are the volume of the central compartment `Central` and the clearance rate `CL_Central`. The PKModelMap property `Estimated` lists these relevant parameters. The underlying algorithm of `sbiofit` assumes parameters are normally distributed, but this assumption may not be true for biological parameters that are constrained to be positive, such as volume and clearance. Specify a log transform for the estimated parameters so that the transformed parameters follow a normal distribution. Use an `estimatedInfo` object to define such transforms and initial values (optional).

```
map.Estimated
```

```
ans = 2x1 cell array  
    {'Central'   }  
    {'CL_Central'}
```

Define such estimated parameters, appropriate transformations, and initial values.

```
estimatedParams = estimatedInfo({'log(Central)', 'log(CL_Central)'}, 'InitialValue', [1 1]);
```

Each infant received a different schedule of dosing. The amount of drug is listed in the data variable `DOSE`. To specify these dosing during fitting, create dose objects from the data. These objects use the property `TargetName` to specify which species in the model receives the dose. In this example, the target species is `Drug_Central`, as listed by the PKModelMap property `Dosed`.

```
map.Dosed
```

```
ans = 1x1 cell array  
    {'Drug_Central'}
```

Create a sample dose with this target name and then use the `createDoses` method of `groupedData` object `data` to generate doses for each infant based on the dosing data `DOSE`.

```
sampleDose = sbiodose('sample', 'TargetName', 'Drug_Central');  
doses = createDoses(data, 'DOSE', '', sampleDose);
```

Fit the model.

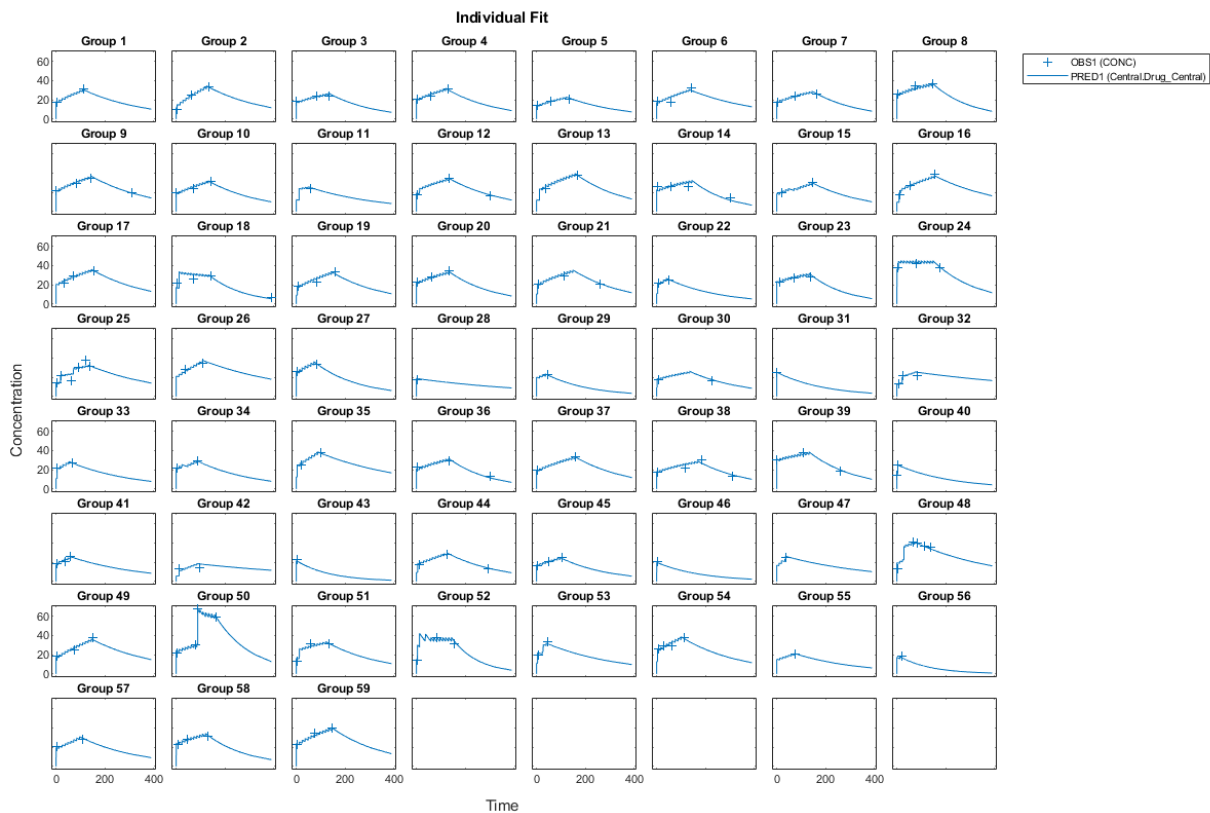
```
[nlmeResults, simI, simP] = sbiofitmixed(onecomp, data, responseMap, estimatedParams, doses,
```

Visualize the fitted results using individual-specific parameter estimates.

```
fig1 = plot(nlmeResults, 'ParameterType', 'individual');
```

```
% Resize the figure.
```

```
fig1.hFig.Position(:) = [100 100 1200 800];
```

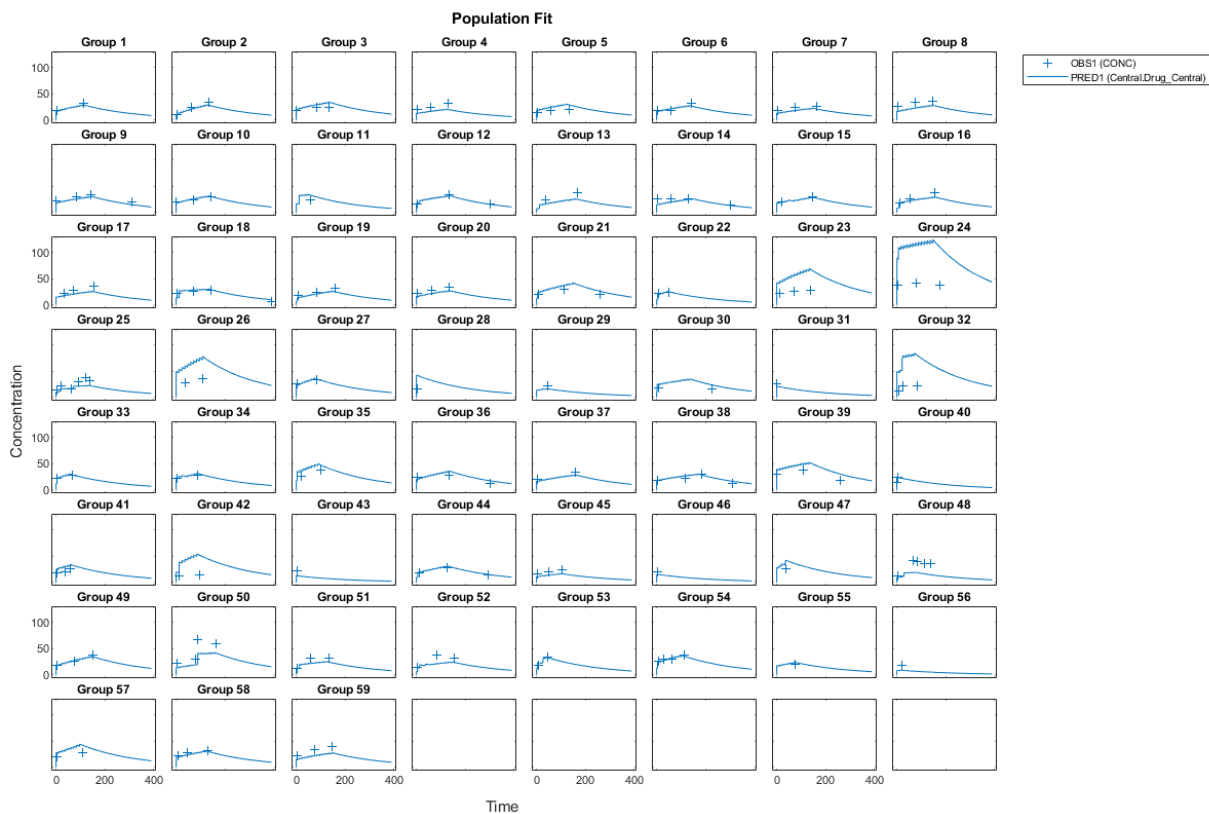


Visualize the fitted results using population parameter estimates.

```
fig2 = plot(nlmeResults, 'ParameterType', 'population');
```

```
% Resize the figure.
```

```
fig2.hFig.Position(:) = [100 100 1200 800];
```



## Input Arguments

### sm — SimBiology model

SimBiology model object

SimBiology model, specified as a SimBiology model object. The active configset object of the model contains solver settings for simulation. Any active doses and variants are applied to the model during simulation unless specified otherwise using the dosing and variants input arguments, respectively.

### grpData — Data to fit

groupedData object

Data to fit, specified as a `groupedData` object.

The name of the time variable must be defined in the `IndependentVariableName` property of `grpData`. For instance, if the time variable name is 'TIME', then specify it as follows.

```
grpData.Properties.IndependentVariableName = 'TIME';
```

`grpData` must have at least two groups, and the name of grouping variable name must be defined in the `GroupVariableName` property of `grpData`. For example, if the grouping variable name is 'GROUP', then specify it as follows.

```
grpData.Properties.GroupVariableName = 'GROUP';
```

A group usually refers to a set of measurements that represent a single time course, often corresponding to a particular individual or experimental condition.

---

**Note** `sbiofitmixed` uses the `categorical` function to identify groups. If any group values are converted to the same value by `categorical`, then those observations are treated as belonging to the same group. For instance, if some observations have no group information (that is, empty character vector), then `categorical` converts empty character vectors to `<undefined>`, and these observations are treated as one group.

---

## **responseMap — Mapping information of model components to response data**

character vector | string | string vector | cell array of character vectors

Mapping information of model components to response data in `grpData`, specified as a character vector, string, string vector, or cell array of character vectors.

Each character vector or string is an equation-like expression, similar to assignment rules in SimBiology. It contains the name (or qualified name) of a quantity (species, compartment, or parameter) in the model `sm`, followed by the character '=' and the name of a variable in `grpData`. For clarity, white spaces are allowed between names and '='.

For example, if you have the concentration data 'CONC' in `grpData` that corresponds to a model species 'Drug\_Central', you can specify the mapping information as follows.

```
responseMap = 'Drug_Central = CONC';
```

To unambiguously name a species, use the qualified name, which includes the name of the compartment. To name a reaction-scoped parameter, use the reaction name to qualify the

parameter. If the name is not a valid MATLAB variable name, surround it by square brackets such as `[reaction 1].[parameter 1]`.

An error is issued if any (qualified) name matches two components of the same type. However, you can use a (qualified) name that matches two components of different types, and the function first finds the species with the given name, followed by compartments and then parameters.

### **covEstiminfo — Estimated parameters**

vector of `estimatedInfo` objects | `CovariateModel` object

Estimated parameters, specified as a vector of `estimatedInfo` objects or a `CovariateModel` object that defines the estimated parameters in the model `sm`, their initial estimates (optional), and their relationship to group-specific covariates included in `grpData` (optional). If this is a vector of `estimatedInfo` objects, then no covariates are used, and all parameters are estimated with group-specific random effects.

You can also specify parameter transformations if necessary. Supported transforms are `log`, `logit`, and `probit`. For details, see `estimatedInfo` object and `CovariateModel` object.

If `covEstiminfo` is a vector of `estimatedInfo` objects, the `CategoryVariableName` property of each of these objects is ignored.

### **dosing — Dosing information**

[] | 2-D matrix of dose objects

Dosing information, specified as an empty array or 2-D matrix of dose objects (`ScheduleDose` object or `RepeatDose` object).

- If empty, no doses are applied during simulation, even if the model has active doses.
- If not empty, the matrix must have a single row or one row per group in the input data. If it has a single row, same doses are applied to all groups during simulation. If it has multiple rows, each row is applied to a separate group, in the same order as the groups appear in the input data.
- Multiple columns are allowed so that you can apply multiple dose objects to each group. All doses in a column must reference the same components in the model. For example, doses in the same column must have the same dose target (`TargetName`). If you parameterize any property of a dose, then all doses within the column must have this property parameterized to the same parameter. If some groups require more doses than others, then fill in the matrix with default (dummy) doses.

- A default dose has default values for all properties, except for the Name property. Create a default dose as follows.

```
d1 = sbiodose('d1');
```

- In addition to manually constructing dose objects using `sbiodose`, if `grpData` has dosing information, you can use the `createDoses` method to construct doses.

## **functionName** — Estimation function name

character vector | string

Estimation function name, specified as a character vector or string. Choices are 'nlmefit' or 'nlmefitsa'. For the summary supported methods and fitting options, see “Supported Methods for Parameter Estimation”.

## **opt** — Options specific to estimation function

struct

Options specific to the estimation function, specified as a structure. The structure contains fields and default values that are the name-value pair arguments accepted by `nlmefit` and `nlmefitsa`, except the following that are not supported.

- 'FEConstDesign'
- 'FEGroupDesign'
- 'FEObsDesign'
- 'FEParamsSelect'
- 'ParamTransform'
- 'REConstDesign'
- 'REGroupDesign'
- 'REObsDesign'
- 'Vectorization'

'REParamsSelect' is only supported when `covEstiminfo` is a vector of `estimatedInfo` objects.

Use the `statset` function only to set the 'Options' field of the `opt` structure as follows.

```
opt.Options = statset('Display','iter','TolX',1e-3,'TolFun',1e-3);
```



For other supported name-value pair arguments (see `nlmefit` and `nlmefitsa`), set them as follows.

```
opt.ErrorModel = 'proportional';
opt.ApproximationType = 'LME';
```

### **variants — Variants**

`[]` | vector of variant objects

Variants, specified as an empty array or vector of variant objects. If empty, no variants are used even if the model has active variants.

### **tf\_parallel — 'UseParallel' option**

`false` (default) | `true`

'UseParallel' option, specified as `true` or `false`. If `true`, and Parallel Computing Toolbox is available, the function performs parameter estimation in parallel.

### **tf\_progress — 'ProgressPlot' option**

`false` (default) | `true`

'ProgressPlot' option, specified as `true` or `false`. If `true`, a new figure opens containing plots that show the progress of parameter estimation. Specifically, the plots show the values of fixed effects parameters (`theta`), the estimates of the variance parameters, that is, the diagonal elements of the covariance matrix of the random effects ( $\Psi$ ), and the log-likelihood. For details, see “Progress Plot”.

## **Output Arguments**

### **fitResults — Estimation results**

`NLMEResults` object

Estimation results, returned as an `NLMEResults` object.

### **simDataI — Simulation results**

vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects representing simulation results for each group (or individual) using fixed-effect and random-effect estimates (individual-specific parameter estimates).

The states reported in `simDataI` are the states that were included in the `responseMap` input argument as well as any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the SimBiology model `sm`.

## **simDataP — Simulation results**

vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects representing simulation results for each group (or individual) using only fixed-effect estimates (population parameter estimates).

The states reported in `simDataP` are the states that were included in the `responseMap` input argument as well as any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the SimBiology model `sm`.

## **References**

- [1] Grasela Jr, T.H., Donn, S.M. (1985) Neonatal population pharmacokinetics of phenobarbital derived from routine clinical data. *Dev Pharmacol Ther.* 8(6), 374-83.

# **Extended Capabilities**

## **Automatic Parallel Support**

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the `'UseParallel'` name-value pair argument.

## **See Also**

`CovariateModel` object | `NLMEResults` object | `estimatedInfo` object | `groupedData` | `nlmefit` | `nlmefitsa` | `sbiofit` | `sbiofitstatusplot`

## **Topics**

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”

“What Is a Nonlinear Mixed-Effects Model?”

“Nonlinear Mixed-Effects Modeling Workflow”

“Specify a Covariate Model”

“Specify an Error Model”

“Maximum Likelihood Estimation”

“Obtain the Fitting Status”

“Supported Methods for Parameter Estimation”

“Progress Plot”

**Introduced in R2014a**

## sbiofitstatusplot

Plot status of nonlinear mixed-effects estimation

### Syntax

```
stop = sbiofitstatusplot(beta, status, state)
stop = sbiofitstatusplot(beta, status, state, fenames)
```

### Description

`stop = sbiofitstatusplot(beta, status, state)` initializes or updates a plot with the fixed effects, *beta*, the log likelihood *status.fval*, and the variance of the random effects, `diag(status.Psi)`.

The function returns an output (*stop*) to satisfy requirements for the 'OutputFcn' option of `nlmefit` or `nlmefitsa`. For `sbiofitstatusplot`, the value of *stop* is always `false`.

`stop = sbiofitstatusplot(beta, status, state, fenames)` specifies the names for the fixed-effects *fenames*.

Use `sbiofitstatusplot` to obtain status information about NLME fitting when using the `sbiofitmixed` function. Specify `@sbiofitstatusplot` for the 'OutputFcn' field of a `statset` option structure and then pass in the structure as an input argument to `sbiofitmixed`.

Alternatively, you can set the 'ProgressPlot' name-value pair argument to `true` when you run `sbiofitmixed`. The function `sbiofitmixed` then calls `sbiofitstatusplot` at each function iteration. For details, see "Progress Plot".

### Input Arguments

#### **beta**

Current fixed effects

**status**

Structure containing several fields

Field	Value
inner	Structure describing the current status of the inner iterations within the ALT and LAP procedures, with the fields: <ul style="list-style-type: none"> <li>procedure <ul style="list-style-type: none"> <li>'PNLS', 'LME', or 'none' when the procedure is 'ALT'</li> <li>'PNLS', 'PLM', or 'none' when the procedure is 'LAP'</li> </ul> </li> <li>state — 'init', 'iter', 'done', or 'none'</li> <li>iteration — Integer starting from 0, or NaN</li> </ul>
procedure	'ALT' or 'LAP'
iteration	Integer starting from 0
fval	Current log-likelihood
Psi	Current random-effects covariance matrix
theta	Current parameterization of Psi
mse	Current error variance

**state**

Either 'init', 'iter', or 'done'.

**fenames**

Character vector, string, string vector, or cell array of character vectors specifying the names of fixed effects

**Examples**

Obtain status information for NLME fitting:

```
% Create a statset option with 'OutputFcn'.
fitOptions.Options = statset('OutputFcn',@sbiofitstatusplot);
% Pass the structure to sbiofitmixed function.
results = sbiofitmixed(..., fitOptions);
```

## **Definitions**

### **Alt**

Alternating algorithm for the optimization of the LME or RELME approximations

### **FO**

First-order estimate

### **FOCE**

First-order conditional estimate

### **LAP**

Optimization of the Laplacian approximation for FO or FOCE

### **LME**

Linear mixed-effects estimation

### **NLME**

Nonlinear mixed effects

### **PLM**

Profiled likelihood maximization

### **PNLS**

Penalized nonlinear least squares

## **RELME**

Restricted likelihood for the linear mixed-effects model

## **See Also**

`nlmefit` | `sbiofitmixed` | `sbionlinit` | `sbionlmefit` | `sbionlmefitsa`

## **Topics**

“Progress Plot”

**Introduced in R2009b**

## sbiofittool

Open SimBiology desktop for population fitting

---

**Note** `sbiofittool` will be removed in a future release. Use `simbiology` instead to open the SimBiology desktop.

---

### Syntax

```
sbiofittool
```

### Description

`sbiofittool` opens the SimBiology desktop in a state designed for:

- Importing and plotting data for fitting
- Selecting from a library of pharmacokinetic models
- Performing population fit tasks using `sbionlmeft` or `sbionlmeftsa`
- Performing individual fit tasks using `sbionlinfit`

`sbiofittool` opens a simplified configuration of the SimBiology desktop. However, all desktop functionality is available.

If you opened the SimBiology desktop using the `simbiology` function, then `sbiofittool` changes the desktop layout to optimize it for population fitting.

### See Also

`simbiology`

**Introduced in R2011a**



# sbiogetmodel

Get model object that generated simulation data

## Syntax

```
modelObj = sbiogetmodel(simDataObj)
```

## Arguments

<i>simDataObj</i>	SimData object returned by the function <code>sbiosimulate</code> or by <code>sbioensemblerun</code> .
<i>modelObj</i>	Model object associated with the SimData object.

## Description

`modelObj = sbiogetmodel(simDataObj)` returns the SimBiology model (*modelObj*) associated with the results from a simulation run (*simDataObj*). You can use this function to find the model object associated with the specified SimData object when you load a project with several model objects and SimData objects.

If the SimBiology model used to generate the SimData object (*simDataObj*) is not currently loaded, *modelObj* is empty.

## Examples

Retrieve the model object that generated the SimData object.

- 1 Create a model object, simulate, and then return the results as a SimData object.

```
modelObj = sbmlimport('oscillator');
simDataObj = sbiosimulate(modelObj);
```

- 2 Get the model that generated the simulation results.

```
modelObj2 = sbiogetmodel(simDataObj)
SimBiology Model - Oscillator
```

Model Components:

```
Models:          0
Parameters:      0
Reactions:       42
Rules:           0
Species:         23
```

- 3 Check that the two models are the same.

```
modelObj == modelObj2
ans =
     1
```

## See Also

`sbiosimulate`

**Introduced before R2006a**

# sbiolasterror

SimBiology last error message

## Syntax

```
sbiolasterror
diagstruct = sbiolasterror
sbiolasterror([])
sbiolasterror(diagstruct)
```

## Arguments

<i>diagstruct</i>	The diagnostic structure holding Type, Message ID, and Message for the errors.
-------------------	--

## Description

`sbiolasterror` or `diagstruct = sbiolasterror` return a SimBiology diagnostic structure array containing the last error(s) generated by the software. The fields of the diagnostic structure are:

Type	'error'
MessageID	The message ID for the error (for example, 'SimBiology:ConfigSetNameClash')
Message	The error message

`sbiolasterror([])` resets the SimBiology last error so that it will return an empty array until the next SimBiology error is encountered.

`sbiolasterror(diagstruct)` will set the SimBiology last error(s) to those specified in the diagnostic structure (*diagstruct*).

## Examples

This example shows how to use `verify` and `sbiolasterror`.

- 1 Import a model.

```
a = sbmlimport('radiodecay.xml')  
  
SimBiology Model - RadioactiveDecay  
  
Model Components:  
Models:          0  
Parameters:      1  
Reactions:       1  
Rules:           0  
Species:         2
```

- 2 Change the `ReactionRate` of a reaction to make the model invalid.

```
a.reactions(1).reactionrate = 'x*y'  
  
SimBiology Model - RadioactiveDecay  
  
Model Components:  
Models:          0  
Parameters:      1  
Reactions:       1  
Rules:           0  
Species:         2
```

- 3 Use the function `verify` to validate the model.

```
a.verify  
  
??? Error using==>simbio\private\odebuilder>buildPatternSubStrings  
The object y does not resolve on reaction with expression 'x*y'.  
  
Error in ==> sbiogate at 22  
feval(varargin{:});  
  
??? --> Error reported from Expression Validation :  
The object 'y' in reaction 'Reaction1' does not resolve  
to any in-scope species or parameters.  
--> Error reported from Dimensional Analysis :  
Could not resolve species, parameter or model object 'y'  
during dimensional analysis.  
--> Error reported from ODE Compilation:  
Error using==>simbio\private\odebuilder>buildPatternSubStrings  
The object y does not resolve on reaction with expression 'x*y'.
```

- 4 Retrieve the error diagnostic struct.

```
p = sbiolasterror
```

```
p =
```

```
1x3 struct array with fields:
```

```
  Type
```

```
  MessageID
```

```
  Message
```

- 5 Display the first error ID and Message.

```
p(1)
```

```
ans =
```

```
    Type: 'Error'
```

```
  MessageID: 'SimBiology:ReactionObjectDoesNotResolve'
```

```
    Message: 'The object 'y' in reaction 'Reaction1'  
              does not resolve to any in-scope  
              species or parameters.'
```

- 6 Reset the sbiolasterror.

```
sbiolasterror([])
```

```
ans =
```

```
    []
```

- 7 Set sbiolasterror to the diagnostic struct.

```
sbiolasterror(p)
```

```
ans =
```

```
1x3 struct array with fields:
```

```
  Type
```

```
  MessageID
```

```
  Message
```

## See Also

sbiolastwarning | verify

**Topics**

sbioroot on page 1-253

**Introduced in R2006a**

# sbiolastwarning

SimBiology last warning message

## Syntax

```
sbiolastwarning
diagstruct = sbiolastwarning
sbiolastwarning([])
sbiolastwarning(diagstruct)
```

## Arguments

<i>diagstruct</i>	The diagnostic structure holding Type, Message ID, and Message for the warnings.
-------------------	--

## Description

`sbiolastwarning` or `diagstruct = sbiolastwarning` return a SimBiology diagnostic structure array containing the last warnings generated by the software. The fields of the diagnostic structure are:

Type	'warning'
MessageID	The message ID for the warning (for example, 'SimBiology:DANotPerformedReactionRate')
Message	The warning message

`sbiolastwarning([])` resets the SimBiology last warning so that it will return an empty array until the next SimBiology warning is encountered.

`sbiolastwarning(diagstruct)` will set the SimBiology last warnings to those specified in the diagnostic structure (*diagstruct*).

## **See Also**

`sbiolasterror` | `verify`

## **Topics**

`sbiroot` on page 1-253

**Introduced in R2006a**



# sbioloadproject

Load project from file

## Syntax

```
sbioloadproject('projFilename')  
sbioloadproject ('projFilename','variableName')  
sbioloadproject projFilename variableName1 variableName2...  
s = sbioloadproject (...)
```

## Description

`sbioloadproject('projFilename')` loads a SimBiology project from a project file (*projFilename*). If no extension is specified, `sbioloadproject` assumes a default extension of `.sbproj`. Alternatively, the command syntax is `sbioloadproject projFilename`.

`sbioloadproject ('projFilename','variableName')` loads only the variable *variableName* from the project file.

`sbioloadproject projFilename variableName1 variableName2...` loads the specified variables from the project.

`s = sbioloadproject (...)` returns the contents of *projFilename* in a variable `s`. `s` is a struct containing fields matching the variables retrieved from the SimBiology project.

You can display the contents of the project file using the `sbiowhos` command.

## See Also

`sbioaddtolibrary` | `sbioremovefromlibrary` | `sbiosaveproject` | `sbiowhos`

## Topics

`sbiosaveproject` on page 1-272

sbiowhos on page 1-340  
sbioaddtolibrary on page 1-22  
sbioremovefromlibrary on page 1-248

**Introduced in R2006a**

# sbiomodel

Construct model object

## Syntax

```
modelObj = sbiomodel('NameValue')
```

```
modelObj = sbiomodel(...'PropertyName', PropertyValue...)
```

## Arguments

<i>NameValue</i>	Required property to specify a unique name for a model object. Enter a character vector or string.
<i>PropertyName</i>	Property name for a Model object from “Property Summary” on page 1-152.
<i>PropertyValue</i>	Property value. Valid value for the specified property.

## Description

*modelObj* = sbiomodel('NameValue') creates and returns a SimBiology model object (*modelObj*). In the model object, this method assigns a value (*NameValue*) to the property Name.

*modelObj* = sbiomodel(...'PropertyName', *PropertyValue*...) defines optional properties. The name-value pairs can be in any format supported by the function set.

Simulate *modelObj* with the function sbiosimulate.

Add objects to a model object using the methods `addkineticlaw` on page 2-34, `addparameter` on page 2-40, `addreaction` on page 2-51, `addrule` on page 2-57, and `addspecies` on page 2-61.

All SimBiology model objects can be retrieved from the SimBiology root object. A SimBiology model object has its `Parent` property set to the SimBiology root object.

## Method Summary

addcompartment (model, compartment)	Create compartment object
addconfigset (model)	Create configuration set object and add to model object
adddose (model)	Add dose object to model
addevent (model)	Add event object to model object
addparameter (model, kineticlaw)	Create parameter object and add to model or kinetic law object
addreaction (model)	Create reaction object and add to model object
addrule (model)	Create rule object and add to model object
addspecies (model, compartment)	Create species object and add to compartment object within model object
addvariant (model)	Add variant to model
copyobj (any object)	Copy SimBiology object and its children
createSimFunction (model)	Create SimFunction object
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
export (model)	Export SimBiology models for deployment and standalone applications
findUnusedComponents (model)	Find unused species, parameters, and compartments in a model
get (any object)	Get object properties
getadjacencymatrix (model)	Get adjacency matrix from model object
getconfigset (model)	Get configuration set object from model object
getdose (model)	Return SimBiology dose object
getequations	Return system of equations for model object
getstoichmatrix (model)	Get stoichiometry matrix from model object
getvariant (model)	Get variant from model
removeconfigset (model)	Remove configuration set from model
removedose (model)	Add dose object to model
removevariant (model)	Remove variant from model
reorder (model, compartment, kinetic law)	Reorder component lists
set (any object)	Set object properties

## Property Summary

Compartments	Array of compartments in model or compartment
Events	Contain all event objects
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parameters	Array of parameter objects
Parent	Indicate parent object
Reactions	Array of reaction objects
Rules	Array of rules in model object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

- 1 Create a SimBiology model object.

```
modelObj = sbiomodel('cell', 'Tag', 'mymodel');
```

- 2 List all modelObj properties and the current values.

```
get(modelObj)
```

MATLAB returns:

```
Annotation: ''
  Models: [0x1 double]
  Name: 'cell'
  Notes: ''
Parameters: [0x1 double]
  Parent: [1x1 SimBiology.Root]
  Species: [0x1 double]
Reactions: [0x1 double]
  Rules: [0x1 double]
  Tag: 'mymodel'
  Type: 'sbiomodel'
UserData: []
```

- 3** Display a summary of modelObj contents.

```
modelObj
```

```
SimBiology Model - cell
```

```
Model Components:
```

```
Models:          0  
Parameters:      0  
Reactions:       0  
Rules:           0  
Species:         0
```

## See Also

```
addcompartment | addconfigset | addevent | addkineticlaw | addparameter |  
addreaction | addrule | addspecies | copyobj | get | model object | sbioroot |  
sbiosimulate | set
```

**Introduced in R2006a**

## sbionca

Compute noncompartmental analysis (NCA) parameters (requires Statistics and Machine Learning Toolbox)

### Syntax

```
ncaparameters = sbionca(data,opt)
```

### Description

`ncaparameters = sbionca(data,opt)` computes NCA parameters from the concentration-time data. The options object `opt` defines the data columns and other calculation options. `ncaparameters` is a table of calculated NCA parameter values for each group.

### Examples

#### Compute NCA Parameters from Concentration-Time Data

Load a synthetic data set that contains the drug concentration measurements of four individuals after an IV bolus dose.

```
load data1.mat
```

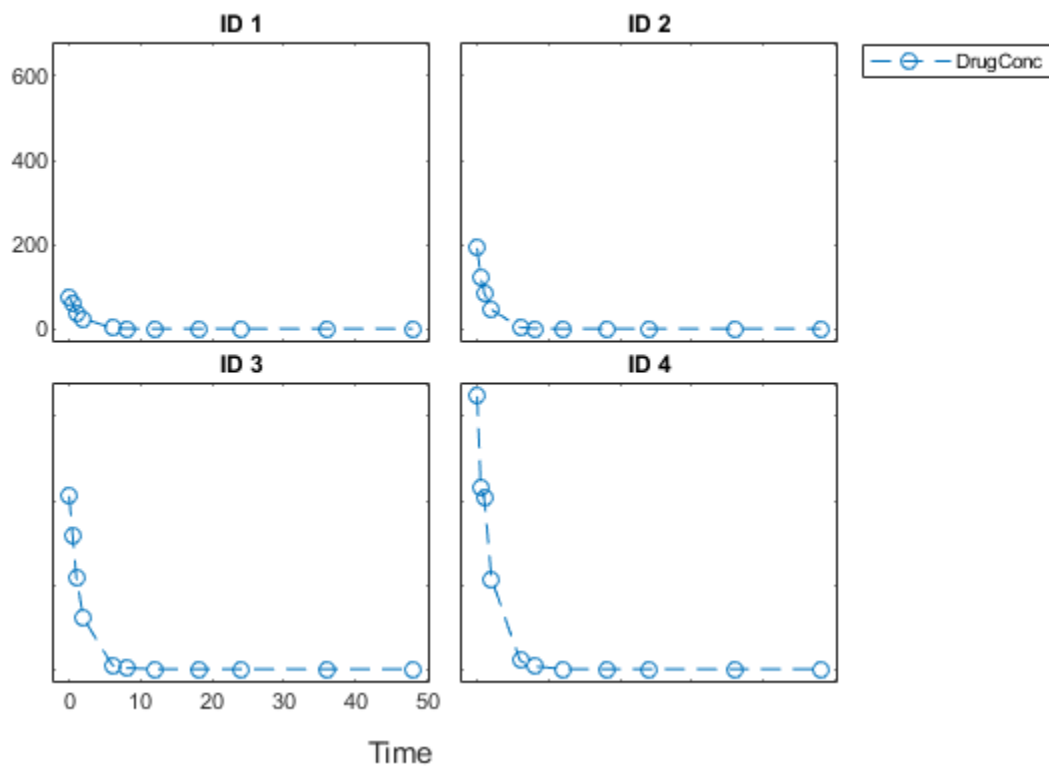
Set the dose amounts to NaN at time points when no dose was administered.

```
data1.Dose(data1.Dose(:) == 0) = NaN;
```

Display the data.

```
sbiotrellis(data1, 'ID', 'Time', 'DrugConc', 'Marker', 'o', 'LineStyle', '--');
```





Categorize the data columns using an NCA options object.

```
opt = sbioncaoptions;
opt.groupColumnName      = 'ID';
opt.concentrationColumnName = 'DrugConc';
opt.timeColumnName      = 'Time';
opt.IVDoseColumnName    = 'Dose';
```

Compute NCA parameters for each individual.

```
ncaparameters = sbionca(data1,opt);
```

Display the first few columns of the table. Each row of `ncaparameters` table represents an individual (or group), and each column lists the corresponding NCA parameter value.

```
ncaparameters(:,1:15)
```

```
ans=4x15 table
   ID   doseSchedule   administrationRoute   Lambda_Z   R2   adjusted_R2
   ---   ---           ---           ---           ---   ---
   1     'Single'       'IVBolus'         0.57893    0.99991    0.9999
   2     'Single'       'IVBolus'         0.66798    0.99998    0.99998
   3     'Single'       'IVBolus'         0.62124    0.99999    0.99999
   4     'Single'       'IVBolus'         0.58011    0.99995    0.99995
```

You can also specify a custom time range to compute  $T_{\max}$  and  $C_{\max}$  within that time range, say from time = 0 to 20. You can do so by setting the `C_max_ranges` property as a cell array of two-element row vector.

```
opt.C_max_ranges = {[5.5 20]};
ncaparameters2 = sbionca(data1,opt);
```

The function reports the  $T_{\max}$  and  $C_{\max}$  values within the range by adding two new columns: `T_max_5_5_20` and `C_max_5_5_20`. Note that in the names of these two columns, the last time point is preceded by two consecutive underscores (`__`).

```
ncaparameters2.T_max_5_5__20(:)
```

```
ans = 4x1
```

```
6
6
6
6
```

```
ncaparameters2.C_max_5_5__20(:)
```

```
ans = 4x1
```

```
2.2719
3.0213
10.0233
19.9006
```

Similarly, you can specify a custom time range to compute the partial AUC value for each group.

```
opt.PartialAreas    = {[0 20]};
ncaparameters3     = sbionca(data1,opt);
ncaparameters3.AUC_0__20(:)
```

```
ans = 4×1
103 ×
```

```
0.1436
0.2994
0.7665
1.3017
```

You can also specify multiple time ranges for `C_max_ranges` and `PartialAreas`.

```
opt.C_max_ranges    = {[0 20],[0 10],[0 15]};
opt.PartialAreas    = {[0 12],[0 30]};
ncaparameters4      = sbionca(data1,opt);
```

## Input Arguments

### **data** — Concentration-time data

table | dataset

Concentration-time data for NCA parameter computation, specified as a table or dataset.

Example: `concData`

### **opt** — Options object to define data columns and calculation options

options object

Options object to define data columns and calculation options, specified as an NCA options object. Use `sbioncaoptions` to create this object and set the options.

Example: `ncaopt`

## Output Arguments

### **ncaparameters** — Calculated NCA parameters

table

Calculated NCA parameters, returned as a table. For details on how the parameters are computed, see “Noncompartmental Analysis”.

## **See Also**

`sbioncaoptions`

## **Topics**

“Noncompartmental Analysis”

**Introduced in R2017b**

# sbioncaoptions

Specify options to calculate noncompartmental analysis (NCA) parameters

## Syntax

```
opt = sbioncaoptions
```

## Description

`opt = sbioncaoptions` returns an NCA options object. Use dot notation to set the object properties for the options.

## Examples

### Compute NCA Parameters from Concentration-Time Data

Load a synthetic data set that contains the drug concentration measurements of four individuals after an IV bolus dose.

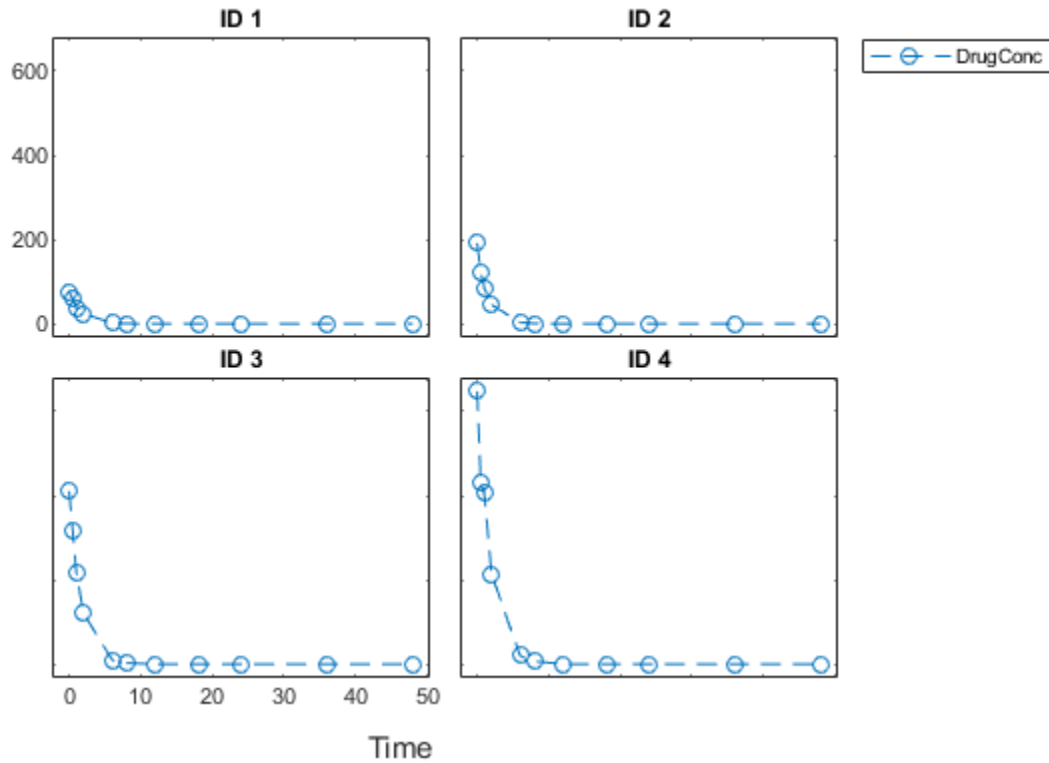
```
load data1.mat
```

Set the dose amounts to NaN at time points when no dose was administered.

```
data1.Dose(data1.Dose(:) == 0) = NaN;
```

Display the data.

```
sbiotrellis(data1, 'ID', 'Time', 'DrugConc', 'Marker', 'o', 'LineStyle', '--');
```



Categorize the data columns using an NCA options object.

```
opt = sbioncaoptions;
opt.groupColumnName      = 'ID';
opt.concentrationColumnName = 'DrugConc';
opt.timeColumnName      = 'Time';
opt.IVDoseColumnName    = 'Dose';
```

Compute NCA parameters for each individual.

```
ncaparameters = sbionca(data1,opt);
```

Display the first few columns of the table. Each row of `ncaparameters` table represents an individual (or group), and each column lists the corresponding NCA parameter value.

```
ncaparameters(:,1:15)
```

```
ans=4×15 table
   ID  doseSchedule  administrationRoute  Lambda_Z  R2  adjusted_R2
   ---  ---  ---  ---  ---  ---
   1    'Single'      'IVBolus'      0.57893  0.99991  0.9999
   2    'Single'      'IVBolus'      0.66798  0.99998  0.99998
   3    'Single'      'IVBolus'      0.62124  0.99999  0.99999
   4    'Single'      'IVBolus'      0.58011  0.99995  0.99995
```

You can also specify a custom time range to compute  $T_{\max}$  and  $C_{\max}$  within that time range, say from time = 0 to 20. You can do so by setting the `C_max_ranges` property as a cell array of two-element row vector.

```
opt.C_max_ranges = {[5.5 20]};
ncaparameters2 = sbionca(data1,opt);
```

The function reports the  $T_{\max}$  and  $C_{\max}$  values within the range by adding two new columns: `T_max_5_5_20` and `C_max_5_5_20`. Note that in the names of these two columns, the last time point is preceded by two consecutive underscores (`__`).

```
ncaparameters2.T_max_5_5__20(:)
```

```
ans = 4×1
```

```
6
6
6
6
```

```
ncaparameters2.C_max_5_5__20(:)
```

```
ans = 4×1
```

```
2.2719
3.0213
10.0233
19.9006
```

Similarly, you can specify a custom time range to compute the partial AUC value for each group.

```
opt.PartialAreas    = {[0 20]};  
ncaparameters3     = sbionca(data1,opt);  
ncaparameters3.AUC_0__20(:)
```

```
ans = 4×1  
103 ×
```

```
0.1436  
0.2994  
0.7665  
1.3017
```

You can also specify multiple time ranges for `C_max_ranges` and `PartialAreas`.

```
opt.C_max_ranges    = {[0 20],[0 10],[0 15]};  
opt.PartialAreas    = {[0 12],[0 30]};  
ncaparameters4      = sbionca(data1,opt);
```

## Output Arguments

### **opt** — Options to calculate NCA parameters

NCA options object

Options to calculate NCA parameters, returned as an NCA options object. The properties of the object are classified into two groups, data classification options and parameter calculation options.



## Data Classification Options

Property	Description
IVDoseColumnName	Name of the data column that contains the IV dose amount.
EVDoseColumnName	Name of the data column that contains the extravascular (EV) dose amount.
concentrationColumnName	Name of the data column that contains the measured concentrations.
timeColumnName	Name of the data column that contains the time points.
groupColumnName	<p>Name of the data column that contains the grouping information. You can specify grouping using two levels of hierarchy. Specify the outer level of grouping in this column. Specify the inner level of grouping (subgroups) in <code>idColumnName</code>.</p> <p>If you specify <code>idColumnName</code>, then you must also specify <code>groupColumnName</code>.</p> <p>For example, consider data that contains three groups, where each group contains four patients. The group column labels the three groups, and the ID column labels each patient.</p>
idColumnName	<p>Name of the data column that contains the grouping information. You can specify grouping using two levels of hierarchy. Specify the inner level of grouping (subgroups) in this column. Specify the outer level of grouping in <code>groupColumnName</code>.</p> <p>If you specify <code>idColumnName</code>, then you must also specify <code>groupColumnName</code>.</p>
infusionRateColumnName	Name of the data column that contains the infusion rates.

**Parameter Calculation Options**

<b>Property</b>	<b>Description</b>
LOQ	Lower limit of quantization, a threshold below which the values of dependent variable are truncated to zero.
AdministrationRoute	Drug administration route. Three types of administration are supported: IVBolus, IVInfusion, and ExtraVascular.
TAU	Dosing interval for multiple-dosing data.
SparseData	Boolean that indicates whether or not the values of dependent variable are averaged between subgroups to further populate a profile for a group. Time values for each measurement across subgroups (IDs) within a group must be identical.
Lambda_Z_Time_Min_Max	Two-element row vector that specifies a custom time range to compute the terminal rate constant (Lambda_z). The time range applies to all groups; you cannot specify a different time range for each group. For details, see “Noncompartmental Analysis”.
PartialAreas	Cell array of one or more two-element row vectors that specify one or more time ranges used to compute the partial AUC values. You can specify multiple rows for group-specific ranges, where the number of rows equal the number of groups. If there is only one row, the same time ranges are used for all groups.

Property	Description
C_max_ranges	Cell array of one or more two-element row vectors that specify one or more time ranges used to report the T_max, C_max pairs within the specified ranges. You can specify multiple rows for group-specific ranges, where the number of rows equal the number of groups. If there is only one row, the same time ranges are used for all groups.

## See Also

sbionca

## Topics

“Noncompartmental Analysis”

**Introduced in R2017b**

## sbionlinfit

Perform nonlinear least-squares regression using SimBiology models (requires Statistics and Machine Learning Toolbox software)

---

**Note** `sbionlinfit` will be removed in a future release. Use `sbiofit` instead.

---

### Syntax

```
results = sbionlinfit(modelObj, pkModelMapObject, pkDataObj,  
InitEstimates)  
results = sbionlinfit(modelObj, pkModelMapObject, pkDataObj,  
InitEstimates, Name, Value)  
results = sbionlinfit(modelObj, pkModelMapObject, pkDataObj,  
InitEstimates, optionStruct)  
[results, SimDataI] = sbionlinfit(...)
```

### Description

*results* = `sbionlinfit(modelObj, pkModelMapObject, pkDataObj, InitEstimates)` performs least-squares regression using the SimBiology model, *modelObj*, and returns estimated results in the *results* structure.

*results* = `sbionlinfit(modelObj, pkModelMapObject, pkDataObj, InitEstimates, Name, Value)` performs least-squares regression, with additional options specified by one or more *Name, Value* pair arguments.

Following is an alternative to the previous syntax:

*results* = `sbionlinfit(modelObj, pkModelMapObject, pkDataObj, InitEstimates, optionStruct)` specifies *optionStruct*, a structure containing fields and values used by the options input structure to the `nlinfit` function.

[*results*, *SimDataI*] = `sbionlinfit(...)` returns simulations of the SimBiology model, *modelObj*, using the estimated values of the parameters.

---

## Input Arguments

### **modelObj**

SimBiology model object used to fit observed data.

---

**Note** If using a model object containing active doses (that is, containing dose objects created using the `adddose` method, and specified as active using the `Active` property of the dose object), be aware that these active doses are ignored by the `sbionlinfit` function.

---

### **pkModelMapObject**

`PKModelMap` object that defines the roles of the model components in the estimation. For details, see `PKModelMap` object.

---

**Note** If using a `PKModelMap` object that specifies multiple doses, ensure each element in the `Dosed` property is unique.

---

### **pkDataObj**

`PKData` object that defines the data to use in fitting, and the roles of the data columns used for estimation. For details, see `PKData` object.

---

**Note** For each subset of data belonging to a single group (as defined in the data column specified by the `GroupLabel` property), the software allows multiple observations made at the same time. If this is true for your data, be aware that:

- These data points are not averaged, but fitted individually.
  - Different numbers of observations at different times cause some time points to be weighted more.
- 

### **InitEstimates**

Vector of initial parameter estimates for each parameter estimated in `pkModelMapObject.Estimated`. The length of `InitEstimates` must equal at least the

length of *pkmodelMObject.Estimated*. The elements of *InitEstimates* are transformed as specified by the *ParamTransform* name-value pair argument.

### **optionStruct**

Structure containing fields and values used by the `options` input structure to the `nlinfit` function. The structure can also use the name-value pairs listed below as fields and values. Defaults for *optionStruct* are the same as for the `options` input structure to `nlinfit`, except for:

- `DerivStep` — Default is the lesser of  $1e-4$ , or the value of the `SolverOptions.RelativeTolerance` property of the configuration set associated with *modelObj*, with a minimum of  $\text{eps}^{(1/3)}$ .
- `FunValCheck` — Default is `off`.

If you have Parallel Computing Toolbox, you can enable parallel computing for faster data fitting by setting the name-value pair argument `'UseParallel'` to `true` in the `statset` options structure as follows:

```
parpool; % Open a parpool for parallel computing
opt = statset(...,'UseParallel',true); % Enable parallel computing
results = sbionlinfit(...,opt); % Perform data fitting
```

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The `Name,Value` arguments are the same as the fields and values in the `options` structure accepted by `nlinfit`. For a complete list, see the `options` input argument in the `nlinfit` reference page in the Statistics and Machine Learning Toolbox documentation. The defaults for `Name,Value` arguments are the same as for the `options` structure accepted by `nlinfit`, except for:

- `DerivStep` — Default is the lesser of  $1e-4$ , or the value of the `SolverOptions.RelativeTolerance` property of the configuration set associated with *modelObj*, with a minimum of  $\text{eps}^{(1/3)}$ .
- `FunValCheck` — Default is `off`.

Following are additional Name, Value arguments that you can use with `sbionlinfit`.

### ParamTransform

Vector of integers specifying a transformation function for each estimated parameter. The transformation function,  $f$ , takes `estimate` as an input and returns `beta`:

$$\text{beta} = f(\text{estimate})$$

Each element in the vector must be one of these integers specifying the transformation for the corresponding value of `estimate`:

- 0 -  $\text{beta} = \text{estimate}$
- 1 -  $\text{beta} = \log(\text{estimate})$  (default)
- 2 -  $\text{beta} = \text{probit}(\text{estimate})$
- 3 -  $\text{beta} = \text{logit}(\text{estimate})$

### ErrorModel

Character vector specifying the form of the error term. Default is 'constant'. Each model defines the error using a standard normal (Gaussian) variable  $e$ , the function value  $f$ , and one or two parameters  $a$  and  $b$ . Choices are:

- 'constant':  $y = f + a*e$
- 'proportional':  $y = f + b*\text{abs}(f)*e$
- 'combined':  $y = f + (a+b*\text{abs}(f))*e$
- 'exponential':  $y = f*\exp(a*e)$ , or equivalently  $\log(y) = \log(f) + a*e$

If you specify an error model, the `results` output argument includes an `errorparam` property, which has the value:

- $a$  for 'constant' and 'exponential'
- $b$  for 'proportional'
- $[a\ b]$  for 'combined'

---

**Note** If you specify an error model, you cannot specify weights.

---

## Weights

Either of the following:

- A matrix of real positive weights, where the number of columns corresponds to the number of responses. That is, the number of columns must equal the number of entries in the `DependentVarLabel` property of `pkDataObj`. The number of rows in the matrix must equal the number of rows in the data set.
- A function handle that accepts a vector of predicted response values and returns a vector of real positive weights.

---

**Note** If using a function handle, the weights must be a function of the response (dependent variable).

---

Default is no weights. If you specify weights, you cannot specify an error model.

## Pooled

Logical specifying whether `sbionlinfit` does fitting for each individual (`false`) or if it pools all individual data and does one fit (`true`). If set to `true`, `sbionlinfit` uses the same model parameters for each dose level.

**Default:** `false`

## Output Arguments

### results

1-by- $N$  array of objects, where  $N$  is the number of groups in `pkDataObj`. There is one object per group, and each object contains these properties:

- `ParameterEstimates` — A dataset array containing fitted coefficients and their standard errors.
- `CovarianceMatrix` — Estimated covariance matrix for the fitted coefficients.
- `beta` — Vector of scalars specifying the fitted coefficients in transformed space.
- `R` — Vector of scalars specifying the residual values, where  $R(i,j)$  is the residual for the  $i$ th time point and the  $j$ th response in the group of data. If your model includes:



- A single response, then R is a column vector of residual values associated with time points in the group of data.
- Multiple responses, then R is a matrix of residual values associated with time points in the group of data, for each response.
- J — Matrix specifying the Jacobian of the model, with respect to an estimated parameter, that is

$$J(i, j, k) = \left. \frac{\partial y_k}{\partial \beta_j} \right|_{t_i}$$

where  $t_i$  is the  $i$ th time point,  $\beta_j$  is the  $j$ th estimated parameter in the transformed space, and  $y_k$  is the  $k$ th response in the group of data.

If your model includes:

- A single response, then J is a matrix of Jacobian values associated with time points in the group of data.
- Multiple responses, then J is a 3-D array of Jacobian values associated with time points in the group of data, for each response.
- COVB — Estimated covariance matrix for the transformed coefficients.
- mse — Scalar specifying the estimate of the error of the variance term.
- errorparam — Estimated parameters of the error model. This property is a scalar if you specify 'constant', 'exponential', or 'proportional' for the error model. This property is a two-element vector if you specify 'combined' for the error model. This property is an empty array if you specify weights using the 'Weights' name-value pair argument.

## SimDataI

SimData object containing data from simulating the model using estimated parameter values for individuals. This object includes observed states and logged states.

## See Also

Model object | PKData object | PKModelDesign object | PKModelDesign object | PKModelMap object | nlinfit | sbionlmefit | sbionlmefitsa

## **Topics**

“Perform Data Fitting with PKPD Models”

**Introduced in R2009a**

# sbionlmeft

Estimate nonlinear mixed effects using SimBiology models (requires Statistics and Machine Learning Toolbox software)

---

**Note** `sbionlmeft` will be removed in a future release. Use `sbiofitmixed` instead.

---

## Syntax

```
results = sbionlmeft(modelObj, pkModelMapObject, pkDataObject,  
InitEstimates)  
results = sbionlmeft(modelObj, pkModelMapObject, pkDataObject,  
CovModelObj)  
results = sbionlmeft(..., Name,Value)  
results = sbionlmeft(..., optionStruct)  
[results, SimDataI, SimDataP] = sbionlmeft(...)
```

## Description

*results* = `sbionlmeft(modelObj, pkModelMapObject, pkDataObject, InitEstimates)` performs nonlinear mixed-effects estimation using the SimBiology model, *modelObj*, and returns estimated results in the *results* structure.

*results* = `sbionlmeft(modelObj, pkModelMapObject, pkDataObject, CovModelObj)` specifies the relationship between parameters and covariates using *CovModelObj*, a `CovariateModel` object. The `CovariateModel` object also provides the parameter transformation.

*results* = `sbionlmeft(..., Name,Value)` performs nonlinear mixed-effects estimation, with additional options specified by one or more *Name,Value* pair arguments.

Following is an alternative to the previous syntax:

*results* = `sbionlmeft(..., optionStruct)` specifies *optionStruct*, a structure containing fields and values, that are the name-value pair arguments accepted by `nlmeft`. The defaults for *optionStruct* are the same as the defaults for the

arguments used by `nlfmfit`, with the exceptions explained in “Input Arguments” on page 1-174.

`[results, SimDataI, SimDataP] = sbionlfmfit(...)` returns simulation data of the SimBiology model, `modelObj`, using the estimated values of the parameters.

## Input Arguments

### **modelObject**

SimBiology model object used to fit observed data.

---

**Note** If using a model object containing active doses (that is, containing dose objects created using the `adddose` method, and specified as active using the `Active` property of the dose object), be aware that these active doses are ignored by the `sbionlfmfit` function.

---

### **pkModelMapObject**

PKModelMap object that defines the roles of the model components used for estimation. For details, see PKModelMap on page 2-279 object.

---

**Note** If using a PKModelMap object that specifies multiple doses, ensure each element in the `Dosed` property is unique.

---

### **pkDataObject**

PKData object that defines the data to use in fitting, and the roles of the columns used for estimation. *pkDataObject* must define target data for at least two groups. For details, see PKData object.

---

**Note** For each subset of data belonging to a single group (as defined in the data column specified by the `GroupLabel` property), the software allows multiple observations made at the same time. If this is true for your data, be aware that:

- These data points are not averaged, but fitted individually.

- Different numbers of observations at different times cause some time points to be weighted more.
- 

### **InitEstimates**

Vector of initial estimates for the fixed effects. The first *P* elements of *InitEstimates* correspond to the fixed effects for each *P* element of *pkModelMapObject.Estimated*. Additional elements correspond to the fixed effects for covariate factors. The first *P* elements of *InitEstimates* are transformed as specified by the *ParamTransform* name-value pairs (log transformed by default).

### **CovModelObj**

*CovariateModel* object that defines the relationship between parameters and covariates. For details, see *CovariateModel* object.

---

**Tip** To simultaneously fit data from multiple dose levels, omit the random effect (*eta*) from the expressions in the *CovariateModel* object.

---

### **optionStruct**

Structure containing fields and values that are the name-value pairs accepted by the *nlmefit* function. The defaults for *optionStruct* are the same as the defaults for the arguments used by *nlmefit*, with the exceptions noted in “Name-Value Pair Arguments” on page 1-176.

If you have Parallel Computing Toolbox, you can enable parallel computing for faster data fitting by setting the name-value pair argument 'UseParallel' to true in the *statset* options structure as follows:

```
parpool; % Open a parpool for parallel computing
opt = statset(...,'UseParallel',true); % Enable parallel computing
results = sbionlmeft(...,'Options',opt); % Perform data fitting
```

---

**Tip** SimBiology software includes the *sbiofitstatusplot* function, which you can specify in the *OutputFcn* field of the *Options* field. This function lets you monitor the status of fitting.

---

**Tip** To simultaneously fit data from multiple dose levels, use the `InitEstimates` input argument and set the value of the `REParamsSelect` field to a 1-by- $n$  logical vector, with all entries set to `false`, where  $n$  equals the number of fixed effects.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

The `sbionlmeFit` function uses the name-value pair arguments supported by the `nlmeFit` function.

These `nlmeFit` name-value pairs are hard-coded in `sbionlmeFit`, and therefore, you cannot set them:

- `FEParamsSelect`
- `FEConstDesign`
- `FEGroupDesign`
- `FEObsDesign`
- `REConstDesign`
- `REGroupDesign`
- `REObsDesign`
- `Vectorization`

If you provide a `CovariateModel` object as input to `sbionlmeFit`, then these `nlmeFit` name-value pairs are computed from the covariate model, and therefore, you cannot set them:

- `FEGroupDesign`
- `ParamTransform`
- `REParamsSelect`

You can set all other `nlmeFit` name-value pairs. For details, see the `nlmeFit` reference page.

---

Be aware that the defaults for these `nImefit` name-value pairs differ when used by `sbionlmeft`:

### **FEGroupDesign**

Numeric array specifying the design matrix for each group.

**Default:** `repmat(eye(P), [1 1 nGroups])`, where `P` = the number of estimated parameters, and `nGroups` = the number of groups in the observed data.

### **ParamTransform**

Vector of integers specifying how the parameters are distributed.

---

**Note** Do not use the `ParamTransform` option to specify parameter transformations when providing a `CovariateModel` object to a fitting function. The `CovariateModel` object provides the parameter transformation.

---

**Default:** Vector of ones, which specifies all parameters are log transformed.

### **OptimFun**

Character vector specifying the optimization function used in maximizing the likelihood.

**Default:** `fminunc`, if you have Optimization Toolbox installed. Otherwise, the default is `fminsearch`.

### **Options**

Structure containing multiple fields, including `DerivStep`, a scalar or vector specifying the relative difference used in the finite difference gradient calculation, and `FunValCheck`, a logical specifying whether to check for invalid values, such as `NaN` or `Inf`, from `modelfun`.

**Default:** The default for `DerivStep` is the lesser of `1e-4`, or the value of the `SolverOptions.RelativeTolerance` property of the configuration set associated with `modelObj`, with a minimum of `eps^(1/3)`. The default for `FunValCheck` is `off`.

---

**Tip** SimBiology software includes the `sbiofitstatusplot` function, which you can specify in the `OutputFcn` field of the `Options` name-value pair input argument. This function lets you monitor the status of fitting.

---

---

**Tip** To simultaneously fit data from multiple dose levels, use the `InitEstimates` input argument and set the `REParamsSelect` name-value pair input argument to a 1-by- $n$  logical vector, with all entries set to `false`, where  $n$  equals the number of fixed effects.

---

## Output Arguments

### results

Structure containing these fields:

- `FixedEffects` — A dataset array containing estimated fixed effects, including standard errors.
- `RandomEffects` — A dataset array containing sampled random effects for each group in the observed data in `pkDataObject`.
- `IndividualParameterEstimates` — A dataset array containing estimated parameter values for individuals, including random effects.
- `PopulationParameterEstimates` — A dataset array containing estimated parameter values for the population, without random effects.
- `RandomEffectCovarianceMatrix` — A dataset array containing the estimated covariance matrix of the random effects.
- `EstimatedParameterNames` — Cell array of character vectors specifying names of the estimated parameters.
- `CovariateNames` — Cell array of character vectors specifying names of the covariates in `CovModelObj`.
- `FixedEffectsStruct` — Structure containing the values of the estimated fixed effects.
- `stats` — Structure containing information such as AIC, BIC, and weighted residuals. For details on the fields in this structure, see the `stats` structure in `nlmefit` in the Statistics and Machine Learning Toolbox documentation. However, the fields in the `stats` structure returned by `sbionlmefit` vary slightly from those returned by `nlmefit`, namely:



- `ires`, `pres`, `iwres`, `pwres`, and `cwres` each contain a matrix of raw or weighted residuals, with the number of columns equal to the number of responses in the model.
- The `stats` structure returned by `sbionlmeft` includes an additional field, `Observed`. This field contains a character vector or cell array of character vectors specifying the measured responses that correspond to the columns in the matrices of the `ires`, `pres`, `iwres`, `pwres`, and `cwres` fields. The `Observed` field is the same as the `Observed` property of the `PKModelMap` input argument.

### **SimDataI**

`SimData` object containing data from simulating the model using the estimated parameter values for individuals. This object includes observed states and logged states.

### **SimDataP**

`SimData` object containing data from simulating the model using the estimated parameter values for the population. This object includes observed states and logged states.

## **See Also**

`Model` object | `PKData` object | `PKModelDesign` object | `PKModelMap` object | `SimData` object | `nlmeft` | `sbiofitstatusplot` | `sbionlinfit` | `sbionlmeftsa`

## **Topics**

“Perform Data Fitting with PKPD Models”

### **Introduced in R2009a**

## sbionlmefitsa

Estimate nonlinear mixed effects with stochastic EM algorithm (requires Statistics and Machine Learning Toolbox software)

---

**Note** `sbionlmefitsa` will be removed in a future release. Use `sbiofitmixed` instead.

---

### Syntax

```
results = sbionlmefitsa(modelObj, pkModelMapObject, pkDataObject,  
InitEstimates)  
results = sbionlmefitsa(modelObj, pkModelMapObject, pkDataObject,  
CovModelObj)  
results = sbionlmefitsa(..., Name, Value)  
results = sbionlmefitsa(..., optionStruct)  
[results, SimDataI, SimDataP] = sbionlmefitsa(...)
```

### Description

*results* = `sbionlmefitsa(modelObj, pkModelMapObject, pkDataObject, InitEstimates)` performs estimations using the Stochastic Approximation Expectation-Maximization (SAEM) algorithm for fitting population data with the SimBiology model, *modelObj*, and returns the estimated results in the *results* structure.

*results* = `sbionlmefitsa(modelObj, pkModelMapObject, pkDataObject, CovModelObj)` specifies the relationship between parameters and covariates using *CovModelObj*, a `CovariateModel` object. The `CovariateModel` object also provides the parameter transformation.

*results* = `sbionlmefitsa(..., Name, Value)` performs estimations using the SAEM algorithm, with additional options specified by one or more *Name, Value* pair arguments.

Following is an alternative to the previous syntax:

*results* = `sbionlmefitsa(..., optionStruct)` specifies *optionStruct*, a structure containing fields and values, that are the name-value pair arguments accepted

by `nlmeftsa`. The defaults for *optionStruct* are the same as the defaults for the name-value pair arguments used by `nlmeftsa`, with the exceptions explained in “Input Arguments” on page 1-181.

`[results, SimDataI, SimDataP] = sbionlmeftsa(...)` returns simulation data of the SimBiology model, *modelObj*, using the estimated values of the parameters.

## Input Arguments

### **modelObject**

SimBiology model object used to fit observed data.

---

**Note** If using a model object containing active doses (that is, containing dose objects created using the `adddose` method, and specified as active using the `Active` property of the dose object), be aware that these active doses are ignored by the `sbionlmeftsa` function.

---

### **pkModelMapObject**

`PKModelMap` object that defines the roles of the model components used for estimation. For details, see `PKModelMap` object.

---

**Note** If using a `PKModelMap` object that specifies multiple doses, ensure each element in the `Dosed` property is unique.

---

### **pkDataObject**

`PKData` object that defines the data to use in fitting and the roles of the columns used for estimation. *pkDataObject* must define target data for at least two groups. For details, see `PKData` object.

---

**Note** For each subset of data belonging to a single group (as defined in the data column specified by the `GroupLabel` property), the software allows multiple observations made at the same time. If this is true for your data, be aware that:

- These data points are not averaged, but fitted individually.
  - Different numbers of observations at different times cause some time points to be weighted more.
- 

### **InitEstimates**

Vector of initial estimates for the fixed effects. The first *P* elements of *InitEstimates* correspond to the fixed effects for each *P* element of *pkModelMapObject.Estimated*. Additional elements correspond to the fixed effects for covariate factors. The first *P* elements of *InitEstimates* are transformed as specified by the *ParamTransform* name-value pair argument (log transformed by default).

### **CovModelObj**

*CovariateModel* object that defines the relationship between parameters and covariates. For details, see *CovariateModel* object.

### **optionStruct**

Structure containing fields and values that are name-value pair arguments accepted by the *nlmefitsa* function. The defaults for *optionStruct* are the same as the defaults for the arguments used by *nlmefitsa*, with the exceptions noted in “Name-Value Pair Arguments” on page 1-183.

If you have Parallel Computing Toolbox, you can enable parallel computing for faster data fitting by setting the name-value pair argument 'UseParallel' to true in the *statset* options structure as follows:

```
parpool; % Open a parpool for parallel computing
opt = statset(...,'UseParallel',true); % Enable parallel computing
results = sbionlmefitsa(...,'Options',opt); % Perform data fitting
```

---

**Tip** SimBiology software includes the *sbiofitstatusplot* function, which you can specify in the *OutputFcn* field of the *Options* field. This function lets you monitor the status of fitting.

---

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

The `sbionlmefitsa` function uses the name-value pair arguments supported by the `nlmefitsa` function.

These `nlmefitsa` name-value pair arguments are hard-coded in `sbionlmefitsa`, and therefore, you cannot set them:

- `FEPARAMSSELECT`
- `FECONSTDESIGN`
- `FEGROUPDESIGN`
- `FE0BSDESIGN`
- `RECONSTDESIGN`
- `REGROUPDESIGN`
- `RE0BSDESIGN`
- `VECTORIZATION`

If you provide a `CovariateModel` object as input to `sbionlmefitsa`, then these `nlmefitsa` name-value pairs are computed from the covariate model, and therefore, you cannot set them:

- `FEGROUPDESIGN`
- `PARAMTRANSFORM`
- `REPARAMSSELECT`

You can set all other `nlmefitsa` name-value pair arguments. For details on these arguments, see the `nlmefitsa` reference page.

Be aware that the defaults for these `nlmefitsa` name-value pair arguments differ when used by `sbionlmefitsa`:

### **FEGroupDesign**

Numeric array specifying the design matrix for each group.

**Default:** `repmat(eye(P), [1 1 nGroups])`, where  $P$  = the number of estimated parameters, and `nGroups` = the number of groups in the observed data.

### **ParamTransform**

Vector of integers specifying how the parameters are distributed.

---

**Note** Do not use the `ParamTransform` option to specify parameter transformations when providing a `CovariateModel` object to a fitting function. The `CovariateModel` object provides the parameter transformation.

---

**Default:** Vector of ones, which specifies all parameters are log transformed.

### **OptimFun**

Character vector specifying the optimization function used in maximizing the likelihood.

**Default:** `fminunc`, if you have Optimization Toolbox installed. Otherwise, the default is `fminsearch`.

### **Options**

Structure containing multiple fields, including `DerivStep`, a scalar or vector specifying the relative difference used in the finite difference gradient calculation, and `FunValCheck`, a logical specifying whether to check for invalid values, such as NaN or Inf, from `modelfun`.

**Default:** The default for `DerivStep` is the lesser of  $1e-4$ , or the value of the `SolverOptions.RelativeTolerance` property of the configuration set associated with `modelObj`, with a minimum of  $\text{eps}^{(1/3)}$ . The default for `FunValCheck` is `off`.

---

**Tip** SimBiology software includes the `sbiofitstatusplot` function, which you can specify in the `OutputFcn` field of the `Options` name-value pair input argument. This function lets you monitor the status of fitting.

---

## Output Arguments

### results

Structure containing these fields:

- `FixedEffects` — A dataset array containing estimated fixed effects, including standard errors.
- `RandomEffects` — A dataset array containing sampled random effects for each group in the observed data in *pkDataObject*.
- `IndividualParameterEstimates` — A dataset array containing estimated parameter values for individuals, including random effects.
- `PopulationParameterEstimates` — A dataset array containing estimated parameter values for the population, without random effects.
- `RandomEffectCovarianceMatrix` — A dataset array containing the estimated covariance matrix of the random effects.
- `EstimatedParameterNames` — Cell array of character vectors specifying names of the estimated parameters.
- `CovariateNames` — Cell array of character vectors specifying names of the covariates in *CovModelObj*.
- `FixedEffectsStruct` — Structure containing the values of the estimated fixed effects.
- `stats` — Structure containing information such as AIC, BIC, and weighted residuals. For details on the fields in this structure, see the `stats` structure in `nlmefitsa` in the Statistics and Machine Learning Toolbox documentation. However, the fields in the `stats` structure returned by `sbionlmefitsa` vary slightly from those returned by `nlmefitsa`, namely:
  - `ires`, `pres`, `iwres`, `pwres`, and `cwres` each contain a matrix of raw or weighted residuals, with the number of columns equal to the number of responses in the model.
  - The `stats` structure returned by `sbionlmeffit` includes an additional field, `Observed`. This field contains a character vector or cell array of character vectors specifying the measured responses that correspond to the columns in the matrices of the `ires`, `pres`, `iwres`, `pwres`, and `cwres` fields. The `Observed` field is the same as the `Observed` property of the `PKModelMap` input argument.

## **SimDataI**

`SimData` object containing data from simulating the model using the estimated parameter values for individuals. This object includes observed states and logged states.

## **SimDataP**

`SimData` object containing data from simulating the model using the estimated parameter values for the population. This object includes observed states and logged states.

## **See Also**

`Model` object | `PKData` object | `PKModelDesign` object | `PKModelMap` object | `SimData` object | `nlmefitsa` | `sbiofitstatusplot` | `sbionlinfit` | `sbionlmefit`

## **Topics**

“Perform Data Fitting with PKPD Models”

## **Introduced in R2010a**



# sbionmfiledef

NONMEM file definition object for sbionmimport

## Syntax

```
nmdefObj = sbionmfiledef  
nmdefObj = sbionmfiledef('PropertyName', PropertyValue)
```

## Description

*nmdefObj* = sbionmfiledef creates a NONMEM® file definition object. The NONMEM file definition object contains properties for specifying the NONMEM data items such as group, time, and dependent variable. The NONMEM file definition object lets you configure the properties to the column heading or the index of the column. Use the NONMEM file definition object in conjunction with the sbionmimport function to import NONMEM formatted files for use in fitting.

*nmdefObj* = sbionmfiledef('PropertyName', PropertyValue) accepts one or more comma-separated property name/value pairs. Specify *PropertyName* inside single quotes. To see the default interpretations for NONMEM formatted files see “Support for Importing NONMEM Formatted Files”.

## Input Arguments

### Filename

If *Filename* extension is .xls or .xlsx it is assumed to be an Excel® file, otherwise it is assumed to be a text file. sbionmfiledef file reads the file using the dataset constructor.

## Property Name/Value Pairs

### **CompartmentLabel**

Identifies the column in the NONMEM formatted file that contains the compartment. Specify the header name as a character vector or specify the index number of the header. During import the `sbionmimport` function uses the information in the column to interpret which compartment receives a dose or measured an observation. The `EventIDLabel` property specifies whether the value is a dose or an observation.

**Default:** ''

### **ContinuousCovariateLabels**

Identifies the column in the NONMEM formatted file that contains continuous covariates. Specify the header name as a character vector or specify the index number of the header.

**Default:** {}

### **DateLabel**

Identifies the column in the NONMEM formatted file that contains the date. Specify the header name as a character vector or specify the index number of the header. During import the `sbionmimport` function uses the information in the column to interpret time information for each dose, response and covariate measurement.

**Default:** ''

### **DependentVariableLabel**

Identifies the column in the NONMEM formatted file that contains observations. Specify the header name as a character vector or specify the index number of the header.

**Default:** ''

### **DoseLabel**

Identifies the column in the NONMEM formatted file that contains the dosing information. Specify the header name as a character vector or specify the index number of the header.

**Default:** ''

**DoseIntervalLabel**

Identifies the column in the NONMEM formatted file that contains the time between doses. Specify the header name as a character vector or specify the index number of the header.

**Default:** ' '

**DoseRepeatLabel**

Identifies the column in the NONMEM formatted file that contains the number of times (excluding the initial dose) that the dose is repeated. Specify the header name as a character vector or specify the index number of the header.

**Default:** ' '

**EventIDLabel**

Identifies the column in the NONMEM formatted file that contains the event identification specifying whether the value is a dose, observation, or covariate. Specify the header name as a character vector or specify the index number of the header.

**Default:** ' '

**GroupLabel**

Identifies the column in the NONMEM formatted file that contains the Group ID. Specify the header name as a character vector or specify the index number of the header.

**Default:** ' '

**MissingDependentVariableLabel**

Identifies the column in the NONMEM formatted file that contains information about whether a row contains an observation event (0), or not (1). Specify the header name as a character vector or specify the index number of the header.

**Default:** ' '

**RateLabel**

Identifies the column in the NONMEM formatted file that contains the rate of infusion. Specify the header name as a character vector or specify the index number of the header.

**Default:** ''

### **TimeLabel**

Identifies the column in the NONMEM formatted file that contains the time or date of observation. During import the `sbionmimport` function uses this information to interpret when a dose was given, an observation or covariate measurement recorded. Specify the header name as a character vector or specify the index number of the header.

**Default:** ''

### **Type**

Identifies the object as 'NMFileDef', (Read-only).

## **Output Arguments**

### **nmdefObj**

Defines the meanings of the file column headings. It contains properties for specifying data items such as group, time and date. `TimeLabel` and `DependentVariableLabel` must be specified.

## **Examples**

Configure a NONMEM file definition object and import data from a NONMEM formatted file.

```
% Configure a NMFileDef object.
def = sbionmfiledef;
def.CompartmentLabel      = 'CPT';
def.DoseLabel             = 'AMT';
def.DoseIntervalLabel    = 'II';
def.DoseRepeatLabel      = 'ADDL';
def.GroupLabel           = 'ID';
def.TimeLabel            = 'TIME';
def.DependentVariableLabel = 'DV';
def.EventIDLabel        = 'EVID';

filename = 'C:\work\datafiles\dose.xls';
ds = sbionmimport(filename, def);
```

## Tips

- Use `sbionmfiledef` with `sbionmimport` if you want to apply NONMEM interpretation of headers, and the data file has column header labels different from the table shown in “Support for Importing NONMEM Formatted Files”.
- Use `sbionmimport` if the data file has column header labels identical to the table shown in “Support for Importing NONMEM Formatted Files”.

## See Also

`sbionmimport`

## Topics

“Importing Data”

**Introduced in R2010a**

## sbionmimport

Import NONMEM-formatted data

### Syntax

```
data = sbionmimport('Filename')  
data = sbionmimport(nmds)  
data = sbionmimport('Filename', nmdefObj)  
data = sbionmimport(_, 'ParameterName', ParameterValue)  
data = sbionmimport(nmds, nmdefObj)  
[data, PKDataObj] = sbionmimport(_)
```

### Description

*data* = sbionmimport('Filename') or *data* = sbionmimport(*nmds*) converts a NONMEM formatted file, and assumes that the file is configured to use the following default values for column headers: ADDL, AMT, CMT, DATE, DV, EVID, ID, II, MDV, RATE, TIME. See “Support for Importing NONMEM Formatted Files” for more information on each of the headers.

*data* = sbionmimport('Filename', *nmdefObj*) imports a NONMEM formatted file named *Filename*, into a SimBiology formatted dataset *data* using the meanings of the file column headings defined in the NONMEM file definition object *nmdefObj*.

*data* = sbionmimport(\_, 'ParameterName', *ParameterValue*) accepts one or more comma-separated name-value pairs that are accepted by the `readtable` method. If additional information is required to read the file such as the delimiter, specify required name-value pairs. See `readtable` for a list of supported name-value pairs.

*data* = sbionmimport(*nmds*, *nmdefObj*) reads a NONMEM formatted dataset *nmds* and returns a `groupedData` object *data*. Each variable in *nmds* must be a column vector.

[*data*, *PKDataObj*] = sbionmimport(\_) returns a `PKData` object, *PKDataObj* containing the dataset *data*. The *PKDataObj* properties show the labels specified in *data*.

## Input Arguments

### Filename

If extension of *Filename* is `.xls` or `.xlsx`, `sbionmimport` assumes it to be an Excel file. Otherwise `sbionmimport` assumes *Filename* is a text file. `sbionmimport` reads the file using `dataset` or `readtable`.

### nmds

NONMEM-formatted data, specified as a `dataset`, `table`, or `groupedData` object. Each variable in `nmds` must be a column vector.

### nmdefObj

*nmdefObj* defines the meanings of the file column headings. *nmdefObj* is a NONMEM file definition object created using the `sbionmfiledef` function. It contains properties for specifying data items such as group, time, and date. You must specify the `TimeLabel` and the `DependentVariableLabel` properties.

When this argument is omitted or empty `[]`, the default NONMEM interpretation is used.

## Output Arguments

### data

`groupedData` object. It contains a separate column for each dose and observation. The `Description` property of *data* contains a list of warnings, if any, that occurred while constructing *data*. To view the warnings, enter the following in the command line.

```
data.Properties.Description
```

### PkDataObj

The `PKData` object defines the data to use in fitting and the roles of the columns used for estimation. For more information, see `PKData` object.

## Examples

## Import a Dataset

Load a sample dataset.

```
load pheno ds;
```

The dataset contains 6 variables (columns). Display the names of these variables.

```
ds.Properties.VariableNames
```

```
ans = 1x6 cell array
      {'ID'}      {'TIME'}      {'DOSE'}      {'WEIGHT'}      {'APGAR'}      {'CONC'}
```

Define what these variables mean according to the NONMEM definition.

```
def = sbionmfiledef;
def.GroupLabel = 'ID';
def.TimeLabel = 'TIME';
def.DependentVariableLabel = 'CONC';
def.DoseLabel = 'DOSE'
```

```
def =
  NMFileDef with properties:
      CompartmentLabel: ''
  ContinuousCovariateLabels: {}
      DateLabel: ''
  DependentVariableLabel: 'CONC'
      DoseLabel: 'DOSE'
  DoseIntervalLabel: ''
  DoseRepeatLabel: ''
  EventIDLabel: ''
      GroupLabel: 'ID'
  MissingDependentVariableLabel: ''
      RateLabel: ''
  TimeLabel: 'TIME'
      Type: 'NMFileDef'
```

```
def.ContinuousCovariateLabels = {'WEIGHT', 'APGAR'};
```

Import the dataset.

```
data = sbionmimport(ds,def);
```



## Import Data from a GroupedData object

Load a sample dataset.

```
load pheno ds
```

Create a groupedData object.

```
grpData = groupedData(ds);
```

Use the groupedData object variable names and define what column headings or variables mean according to the NONMEM definition.

```
def = sbionmfiledef;  
def.GroupLabel = grpData.Properties.GroupVariableName;  
def.TimeLabel = grpData.Properties.IndependentVariableName;  
def.DependentVariableLabel = 'CONC';  
def.DoseLabel = 'DOSE';  
def.ContinuousCovariateLabels = {'WEIGHT', 'APGAR'};
```

Import the dataset.

```
data = sbionmimport(grpData,def);
```

## See Also

sbionmfiledef

## Topics

“Importing Data”

**Introduced in R2010a**

## sbioparameterci

Compute confidence intervals for estimated parameters (requires Statistics and Machine Learning Toolbox)

### Syntax

```
ci = sbioparameterci(fitResults)
ci = sbioparameterci(fitResults,Name,Value)
```

### Description

`ci = sbioparameterci(fitResults)` computes 95% confidence intervals for the estimated parameters from `fitResults`, an `NLINResults` object or `OptimResults` object returned by the `sbiofit` function. `ci` is a `ParameterConfidenceInterval` object that contains the computed confidence intervals.

`ci = sbioparameterci(fitResults,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Examples

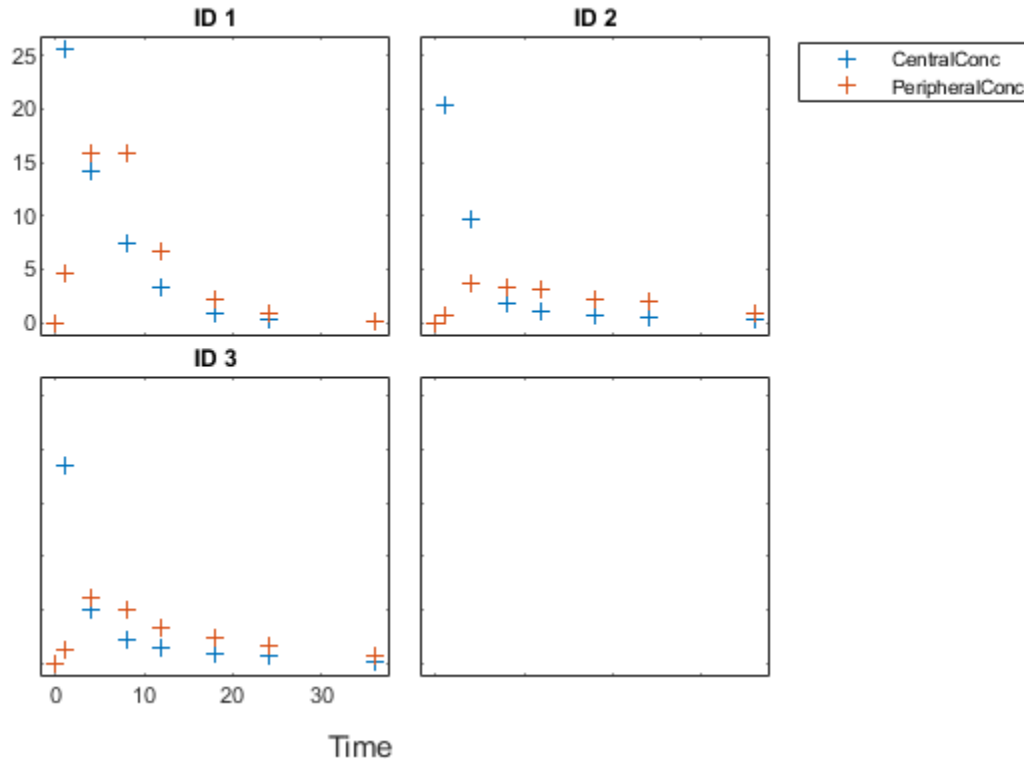
#### Compute Confidence Intervals for Estimated PK Parameters and Model Predictions

##### Load Data

Load the sample data to fit. The data is stored as a table with variables *ID*, *Time*, *CentralConc*, and *PeripheralConc*. This synthetic data represents the time course of plasma concentrations measured at eight different time points for both central and peripheral compartments after an infusion dose for three individuals.

```
clear all
load data10_32R.mat
gData = groupedData(data);
```

```
gData.Properties.VariableUnits = {'', 'hour', 'milligram/liter', 'milligram/liter'};
sbiotrellis(gData, 'ID', 'Time', {'CentralConc', 'PeripheralConc'}, 'Marker', '+', ...
            'LineStyle', 'none');
```



## Create Model

Create a two-compartment model.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2 = addCompartment(pkmd, 'Peripheral');
```

```
model          = construct(pkmd);  
configset     = getconfigset(model);  
configset.CompileOptions.UnitConversion = true;
```

### Define Dosing

Define the infusion dose.

```
dose          = sbiodose('dose', 'TargetName', 'Drug_Central');  
dose.StartTime = 0;  
dose.Amount   = 100;  
dose.Rate     = 50;  
dose.AmountUnits = 'milligram';  
dose.TimeUnits  = 'hour';  
dose.RateUnits  = 'milligram/hour';
```

### Define Parameters

Define the parameters to estimate. Set the parameter bounds for each parameter. In addition to these explicit bounds, the parameter transformations (such as log, logit, or probit) impose implicit bounds.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};  
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};  
estimatedParam  = estimatedInfo(paramsToEstimate, ...  
                                'InitialValue', [1 1 1 1], ...  
                                'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

### Fit Model

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

Perform a pooled fit, that is, one set of estimated parameters for all patients.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true);
```

### Compute Confidence Intervals for Estimated Parameters

Compute 95% confidence intervals for each estimated parameter in the unpooled fit.

```
ciParamUnpooled = sbioparameterci(unpooledFit);
```

## Display Results

Display the confidence intervals in a table format. For details about the meaning of each estimation status, see “Parameter Confidence Interval Estimation Status” on page 2-374.

```
ci2table(ciParamUnpooled)
```

```
ans =
```

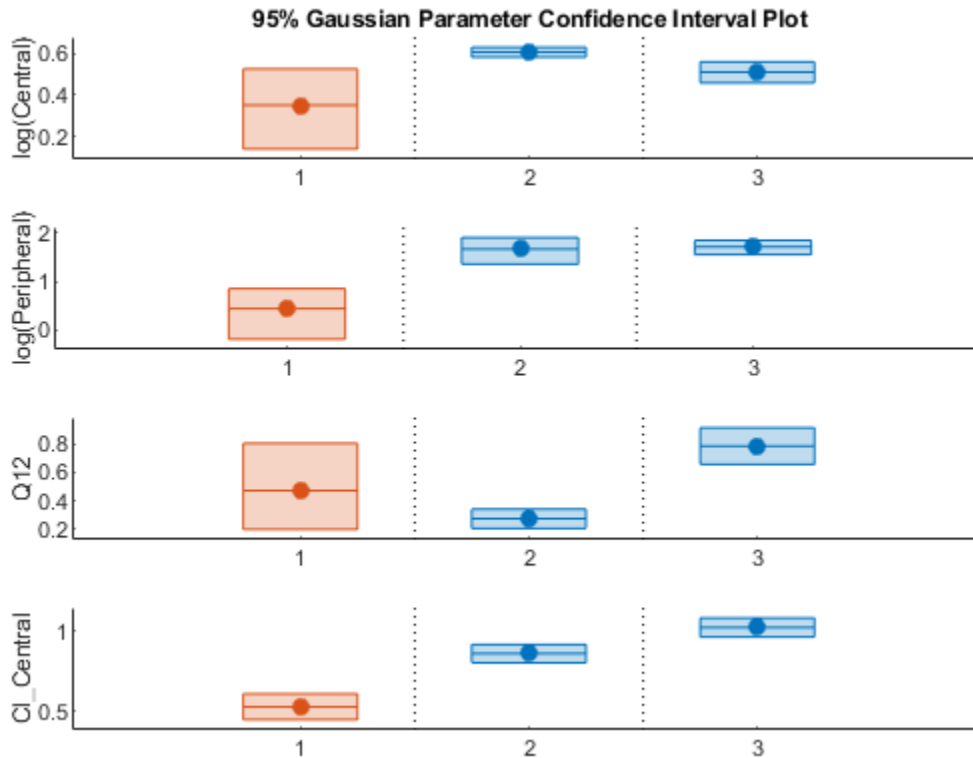
```
12x7 table
```

Group	Name	Estimate	ConfidenceInterval		Type	Alpha
1	'Central'	1.422	1.1533	1.6906	Gaussian	0.05
1	'Peripheral'	1.5629	0.83143	2.3551	Gaussian	0.05
1	'Q12'	0.47159	0.20093	0.80247	Gaussian	0.05
1	'Cl_Central'	0.52898	0.44842	0.60955	Gaussian	0.05
2	'Central'	1.8322	1.7893	1.8751	Gaussian	0.05
2	'Peripheral'	5.3368	3.9133	6.7602	Gaussian	0.05
2	'Q12'	0.27641	0.2093	0.34351	Gaussian	0.05
2	'Cl_Central'	0.86034	0.80313	0.91755	Gaussian	0.05
3	'Central'	1.6657	1.5818	1.7497	Gaussian	0.05
3	'Peripheral'	5.5632	4.7557	6.3708	Gaussian	0.05
3	'Q12'	0.78361	0.65581	0.91142	Gaussian	0.05
3	'Cl_Central'	1.0233	0.96375	1.0828	Gaussian	0.05

Plot the confidence intervals. If the estimation status of a confidence interval is **success**, it is plotted in blue (the first default color). Otherwise, it is plotted in red (the second default color), which indicates that further investigation into the fitted parameters may be required. If the confidence interval is **not estimable**, then the function plots a red line with a centered cross. If there are any transformed parameters with estimated values 0 (for the log transform) and 1 or 0 (for the probit or logit transform), then no confidence intervals are plotted for those parameter estimates. To see the color order, type `get(groot, 'defaultAxesColorOrder')`.

Groups are displayed from left to right in the same order that they appear in the `GroupNames` property of the object, which is used to label the x-axis. The y-labels are the transformed parameter names.

```
plot(ciParamUnpooled)
```



Compute the confidence intervals for the pooled fit.

```
ciParamPooled = sbioparameterci(pooledFit);
```

Display the confidence intervals.

```
ci2table(ciParamPooled)
```

ans =

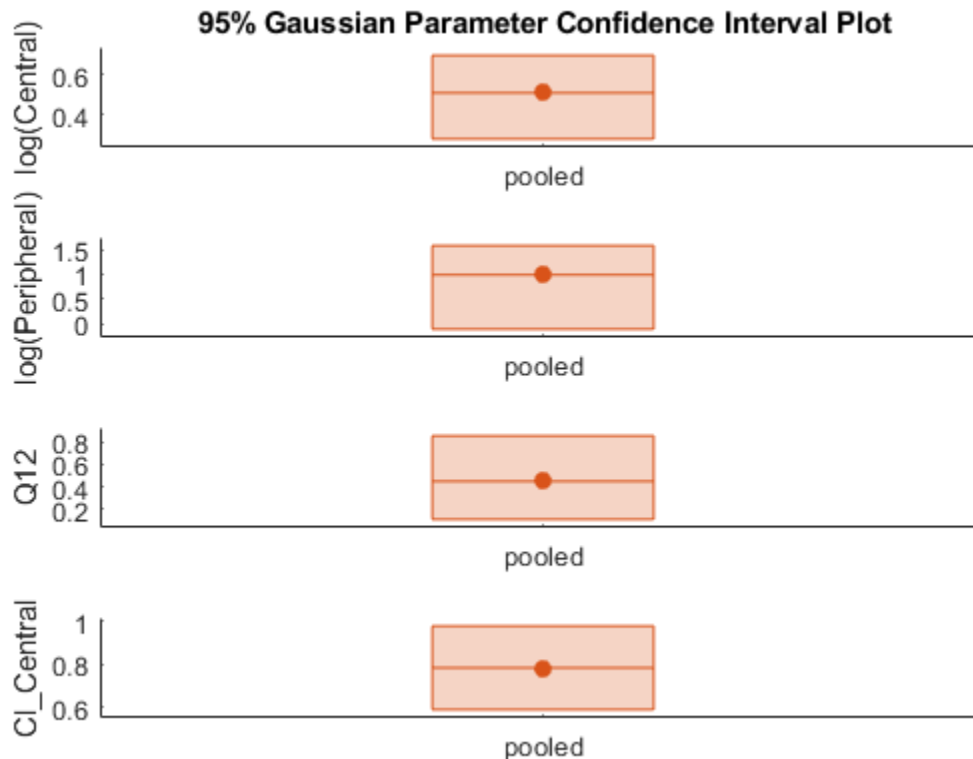
4x7 table

Group	Name	Estimate	ConfidenceInterval	Type	Alpha
-------	------	----------	--------------------	------	-------

pooled	'Central'	1.6626	1.3287	1.9965	Gaussian	0.05	es
pooled	'Peripheral'	2.687	0.89848	4.8323	Gaussian	0.05	co
pooled	'Q12'	0.44956	0.11445	0.85152	Gaussian	0.05	co
pooled	'Cl_Central'	0.78493	0.59222	0.97764	Gaussian	0.05	es

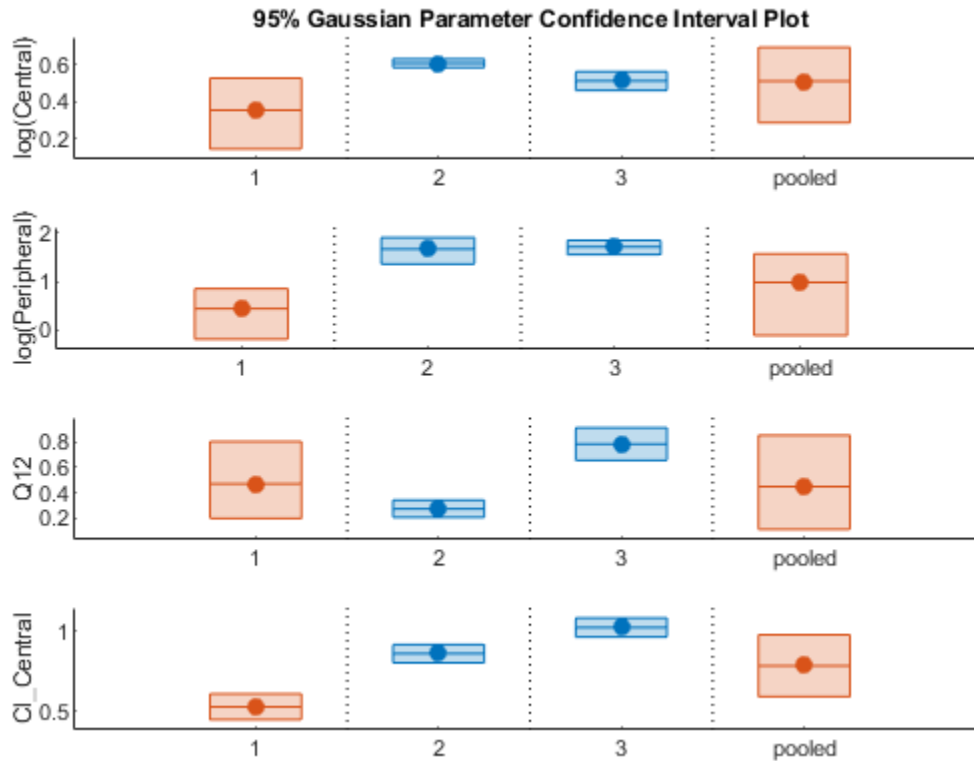
Plot the confidence intervals. The group name is labeled as "pooled" to indicate such fit.

```
plot(ciParamPooled)
```



Plot all the confidence interval results together. By default, the confidence interval for each parameter estimate is plotted on a separate axes. Vertical lines group confidence intervals of parameter estimates that were computed in a common fit.

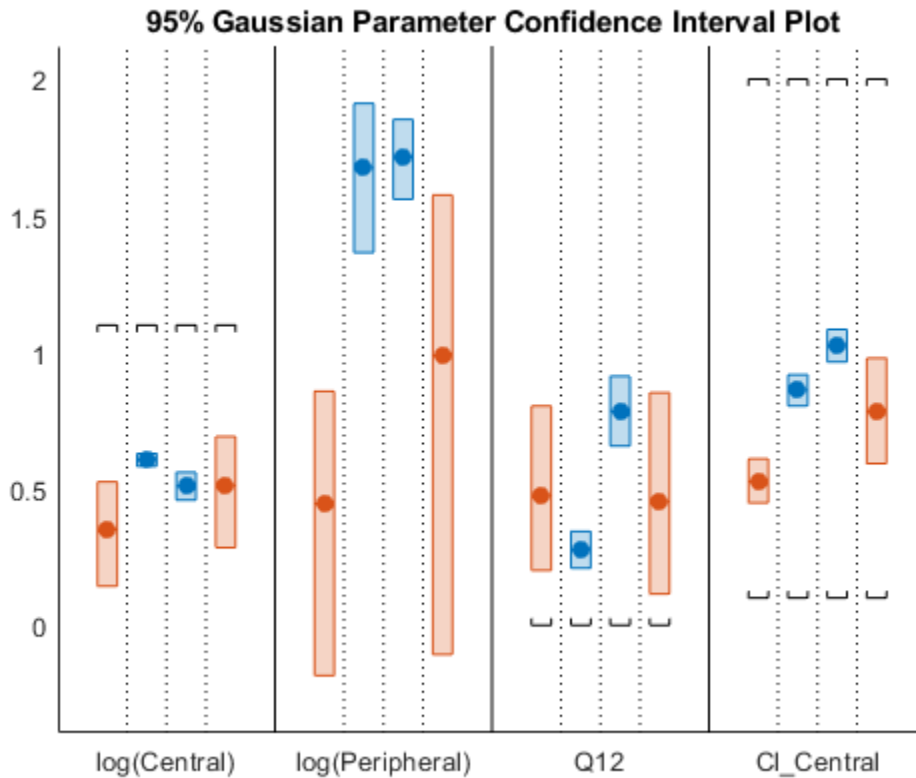
```
ciAll = [ciParamUnpooled;ciParamPooled];  
plot(ciAll)
```



You can also plot all confidence intervals in one axes grouped by parameter estimates using the 'Grouped' layout.

```
plot(ciAll, 'Layout', 'Grouped')
```





In this layout, you can point to the center marker of each confidence interval to see the group name. Each estimated parameter is separated by a vertical black line. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets. Note the different scales on the y-axis due to parameter transformations. For instance, the y-axis of Q12 is in the linear scale, but that of Central is in the log scale due to its log transform.

### Compute Confidence Intervals for Model Predictions

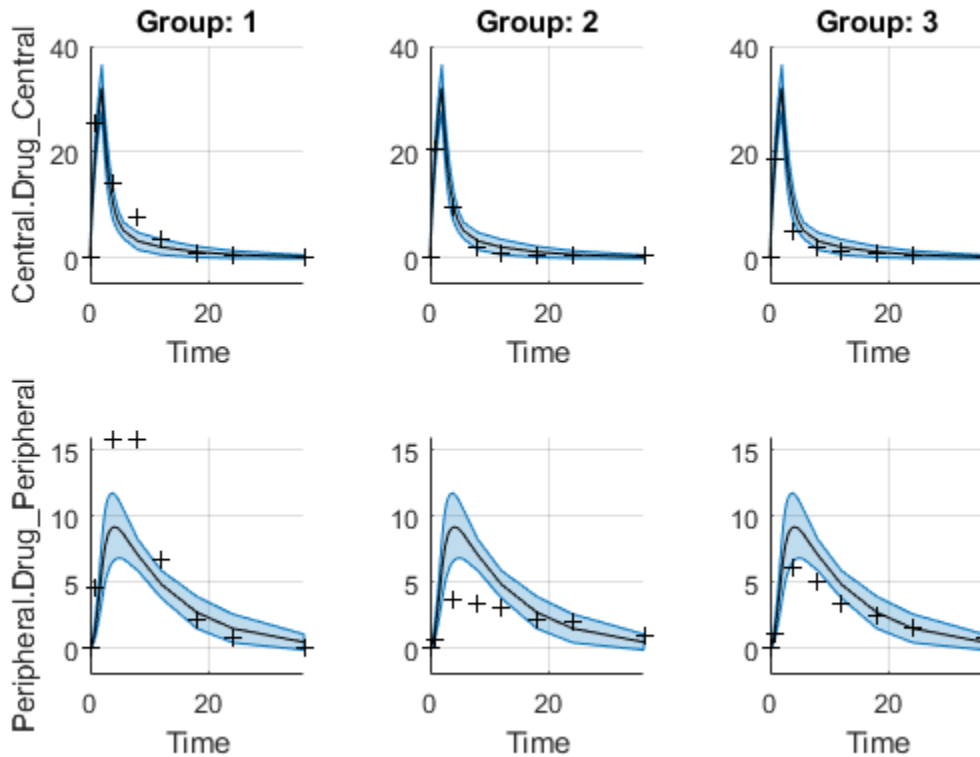
Calculate 95% confidence intervals for the model predictions, that is, simulation results using the estimated parameters.

```
% For the pooled fit
ciPredPooled = sbiopredictionci(pooledFit);
% For the unpooled fit
ciPredUnpooled = sbiopredictionci(unpooledFit);
```

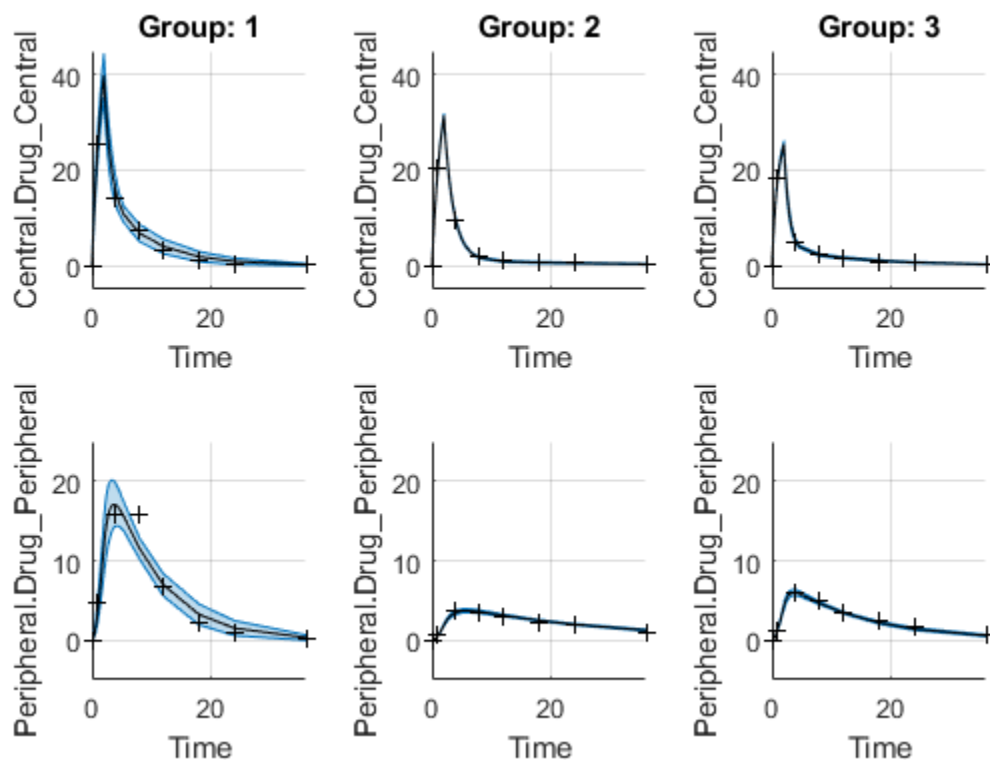
### Plot Confidence Intervals for Model Predictions

The confidence interval for each group is plotted in a separate column, and each response is plotted in a separate row. Confidence intervals limited by the bounds are plotted in red. Confidence intervals not limited by the bounds are plotted in blue.

```
plot(ciPredPooled)
```



```
plot(ciPredUnpooled)
```



## Input Arguments

### **fitResults** – Parameter estimation results from sbiofit

NLINResults object | OptimResults object | vector

Parameter estimation results from `sbiofit`, specified as an `NLINResults` object, `OptimResults` object, or a vector of objects for unpooled fits that were returned from the same `sbiofit` call.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Alpha', 0.01, 'Type', 'profileLikelihood'` specifies to compute a 99% confidence interval using the profile likelihood approach.

### **Alpha — Confidence level**

0.05 (default) | positive scalar

Confidence level,  $(1 - \text{Alpha}) * 100\%$ , specified as the comma-separated pair consisting of `'Alpha'` and a positive scalar between 0 and 1. The default value is `0.05`, meaning a 95% confidence interval is computed.

Example: `'Alpha', 0.01`

### **Type — Confidence interval type**

`'gaussian'` (default) | `'profileLikelihood'` | `'bootstrap'`

Confidence interval type, specified as the comma-separated pair consisting of `'Type'` and a character vector. The valid choices are:

- `'gaussian'` - Use the Gaussian approximation on page 1-209 of the distribution of parameter estimates.
- `'profileLikelihood'` - Compute the profile likelihood intervals. This type is not supported for parameter estimates from hierarchical models, that is, estimated results from fitting different categories on page 1-77 (such as Age or Sex). In other words, if you set the `CategoryVariableName` property of the `EstimatedInfo` object in your original fit, then the fit results are hierarchical, therefore, you cannot compute the `profileLikelihood` confidence intervals on the results. For details, see “Profile Likelihood Confidence Interval Calculation” on page 1-210.
- `'bootstrap'` - Compute confidence intervals using the bootstrap method on page 1-212.

Example: `'Type', 'bootstrap'`

### **Tolerance — Tolerance for profile likelihood and bootstrap confidence interval computations**

1e-5 (default) | positive scalar

Tolerance for the profile likelihood and bootstrap confidence interval computations, specified as the comma-separated pair consisting of 'Tolerance' and a positive scalar.

The profile likelihood method uses this value as a termination tolerance. For details, see "Profile Likelihood Confidence Interval Calculation" on page 1-210.

The bootstrap method uses this value to determine whether a confidence interval is constrained by bounds specified in the original fit. For details, see "Bootstrap Confidence Interval Calculation" on page 1-212.

Example: 'Tolerance',1e-6

### **MaxStepSize — Maximum step size used for computing profile likelihood curves**

0.1 (default) | positive scalar | [] | cell array

Maximum step size used for computing profile likelihood curves, specified as the comma-separated pair consisting of 'MaxStepSize' and a positive scalar, [], or cell array. By default, this argument is set to 0.1. If you set it to [], then the maximum step size is set to 10% of the width of the Gaussian approximation of the confidence interval, if it exists. You can specify a maximum step size (or []) for each estimated parameter using a cell array.

Example: 'MaxStepSize',0.5

### **NumSamples — Number of samples for bootstrapping**

1000 (default) | positive integer

Number of samples for bootstrapping, specified as the comma-separated pair consisting of 'NumSamples' and a positive integer. This number defines the number of fits that are performed during the confidence interval computation to generate bootstrap samples. The smaller the number is, the faster the computation of the confidence intervals becomes, at the cost of decreased accuracy.

Example: 'NumSamples',500

### **Display — Level of display returned to the command line**

'off' (default) | 'none' | 'final' | 'iter'

Level of display returned to the command line, specified as the comma-separated pair consisting of 'Display' and a character vector. 'off' (default) or 'none' displays no output. 'final' displays a message when a computation finishes. 'iter' displays output at each iteration.

Example: 'Display','final'

## **UseParallel** — Logical flag to compute confidence intervals in parallel

true | false

Logical flag to compute confidence intervals in parallel, specified as the comma-separated pair consisting of 'UseParallel' and true or false. By default, the parallel options in the original fit are used. If this argument is set to true and Parallel Computing Toolbox is available, the parallel options in the original fit are ignored, and confidence intervals are computed in parallel.

For the Gaussian confidence intervals:

- If the input `fitResults` is a vector of results objects, then the computation of confidence intervals for each object is performed in parallel. The Gaussian confidence intervals are quick to compute. So, it might be more beneficial to parallelize the original fit (`sbiofit`) and not set `UseParallel` to true for `sbioparameterci`.

For the Profile Likelihood confidence intervals:

- If the number of results objects in the input `fitResults` vector is greater than the number of estimated parameters, then the computation of confidence intervals for each object is performed in parallel.
- Otherwise, the confidence intervals for all estimated parameters within one results object are computed in parallel before the function moves on to the next results object.

For the Bootstrap confidence intervals:

- The function forwards the `UseParallel` flag to `bootci`. There is no parallelization over the input vector of results objects.

---

**Note** If you have a global stream for random number generation with several substreams to compute in parallel in a reproducible fashion, `sbioparameterci` first checks to see if the number of workers is same as the number of substreams. If so, `sbioparameterci` sets `UseSubstreams` to true in the `statset` option and passes it to `bootci`. Otherwise, the substreams are ignored by default.

---

Example: 'UseParallel',true

## Output Arguments

### ci – Confidence interval results

ParameterConfidenceInterval object

Confidence interval results, returned as a ParameterConfidenceInterval object. For an unpooled fit, ci can be a vector of ParameterConfidenceInterval objects.

## Definitions

---

**Note** All confidence interval computations are based on the untransformed parameters. Only when plotted, the confidence intervals are mapped to the transformed space using the parameter transformations defined in the original fit.

---

## Gaussian Confidence Interval Calculation

The function uses the Wald test statistic [1] to compute the confidence intervals. Assuming that there are enough data, the parameter estimates,  $P_{est}$ , are approximately Student's t-distributed with the covariance matrix  $S$  (the CovarianceMatrix property of the results object) returned by sbiofit.

The confidence interval for the  $i$ th parameter estimate  $P_{est,i}$  is computed as follows:

$$P_{est,i} \pm \sqrt{S_{i,i}} * T_{inv} \left( 1 - \frac{Alpha}{2} \right)$$

, where  $T_{inv}$  is the Student's t inverse cumulative distribution function (tinv) with the probability  $1 - (Alpha/2)$ , and  $S_{i,i}$  is the diagonal element (variance) of the covariance matrix  $S$ .

In cases where the confidence interval is constrained by the parameter bounds defined in the original fit, the confidence interval bounds are adjusted according to the approach described by Wu, H. and Neale, M. [2].

- For each parameter estimate, the function first decides whether the confidence interval of the parameter estimate is unbounded. If so, the function sets the estimation status of the corresponding parameter estimate to not estimable.

- Otherwise, if the confidence interval for a parameter estimate is constrained by a parameter bound defined in the original fit, the function sets the estimation status to `constrained`. Parameter transformations (such as `log`, `probit`, or `logit`) impose implicit bounds on the estimated parameters, for example, positivity constraints. Such bounds can lead to the overestimation of confidence, that is, the confidence interval can be smaller than expected.
- If no confidence interval has the estimation status `not estimable` or `constrained`, then the function sets the estimation statuses of all parameter estimates to `success`. Otherwise, the estimation statuses of remaining parameter estimates are set to `estimable`.

## Profile Likelihood Confidence Interval Calculation

Define  $L$  to be the likelihood,  $LH$ , of the parameter estimates (stored in the `ParameterEstimates` property of the results object) returned by `sbiofit`,

$L = LH(P_{est})$ , where  $P_{est}$  is a vector of parameter estimates,  $P_{est,1}, P_{est,2}, \dots, P_{est,n}$ .

The profile likelihood function  $PL$  for a parameter  $P_i$  is defined as:

$$PL(P_i) = \max_{P_j, j \neq i} LH(P_1, \dots, P_i, \dots, P_n), \text{ where } n \text{ is the total number of parameters.}$$

Per Wilks's Theorem [3], the likelihood ratio test statistic,

$$-2 \log \left( \frac{PL(P_i)}{L} \right), \text{ is chi-square distributed with 1 degree of freedom.}$$

$$\log(L) - \log(PL(P_i)) \leq \frac{chiinv(1, 1 - \alpha)}{2}$$

Therefore, find all  $P_i$  so that:

$$\log(PL(P_i)) \geq \log(L) - \frac{chiinv(1, 1 - \alpha)}{2}, \text{ where}$$

Equivalently,

$\log(L) - \frac{chiinv(1, 1 - \alpha)}{2}$  is the target value used in computing the log profile likelihood curve as described next.



- 1 Start at  $P_{est,i}$  and evaluate the likelihood  $L$ .
- 2 Compute the log profile likelihood at  $P_{est,i} + k * \text{MaxStepSize}$  for each side (or direction) of the confidence interval, that is,  $k = 1, 2, 3, \dots$  and  $k = -1, -2, -3, \dots$ .
- 3 Stop if one of these stopping criteria is met on each side.
  - The log profile likelihood falls below the target value. In this case, start bisecting between  $P_{below}$  and  $P_{above}$ , where  $P_{below}$  is the parameter value with the largest log profile likelihood value below the target value, and  $P_{above}$  the parameter value with the smallest log profile likelihood value greater than the target value. Stop the bisection if one of the following is true:
    - Either neighboring log profile likelihood values are less than Tolerance on page 1-0 apart. Set the status for the corresponding side of the confidence interval to **success**.
    - The bisection interval becomes smaller than  $\max(\text{Tolerance}, 2 * \text{eps}('double'))$  and the profile likelihood curve computed so far is above the target value. Set the status of the corresponding side to **not estimable**.
    - The linear gradient approximation of the profile likelihood curve (finite difference between two neighboring parameter values) is larger than -Tolerance on page 1-0 (the negative value of the tolerance). Set the status of the corresponding side to **not estimable**.
  - The step is limited by a bound defined in the original fit. Evaluate at the bound and set the status of the corresponding side to **constrained**.
- If both sides of the confidence interval are unsuccessful, that is, have the status **not estimable**, the function sets the estimation status (`ci.Results.Status` on page 2-0 ) to **not estimable**.
- If no side has the status **not estimable** and one side has the status **constrained**, the function sets the estimation status (`ci.Results.Status` on page 2-0 ) to **constrained**.
- If the computation for all parameters on both sides of the confidence intervals is successful, set the estimation status (`ci.Results.Status` on page 2-0 ) to **success**.
- Otherwise, the function sets the estimation statuses of the remaining parameter estimates to **estimable**.

## Bootstrap Confidence Interval Calculation

The `bootci` function from Statistics and Machine Learning Toolbox is used to compute the bootstrap confidence intervals. The first input `nboot` is the number of samples (`NumSamples`), and the second input `bootfun` is a function that performs these actions:

- Resample the data (independently within each group, if multiple groups are available).
- Run a parameter fit with the resampled data.
- Return the estimated parameters.

If a confidence interval is closer than `Tolerance` to a parameter bound, as defined in the original fit, the function sets the estimation status to `constrained`. If all confidence intervals are further away from the parameter bounds than `Tolerance`, the function sets the status to `success`. Otherwise, it is set to `estimable`.

## References

- [1] Wald, A. "Tests of Statistical Hypotheses Concerning Several Parameters when the Number of Observations is Large." *Transactions of the American Mathematical Society*. 54 (3), 1943, pp. 426-482.
- [2] Wu, H., and M.C. Neale. "Adjusted Confidence Intervals for a Bounded Parameter." *Behavior Genetics*. 42 (6), 2012, pp. 886-898.
- [3] Wilks, S.S. "The Large-Sample Distribution of the Likelihood Ratio for Testing Composite Hypotheses." *The Annals of Mathematical Statistics*. 9 (1), 1938, pp. 60-62.

## Extended Capabilities

### Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the 'UseParallel' name-value pair argument.

## **See Also**

ConfidenceInterval | ParameterConfidenceInterval | sbiofit |  
sbiopredictionci

**Introduced in R2017b**

## sbioparamestim

Perform parameter estimation

---

**Note** `sbioparamestim` will be removed in a future release. Use `sbiofit` instead.

---

---

**Note** Statistics and Machine Learning Toolbox™, Optimization Toolbox™, and Global Optimization Toolbox are recommended for this function.

---

### Syntax

```
[k, result]= sbioparamestim(modelObj, tspan, xtarget,  
observed_array, estimated_array)  
[___]= sbioparamestim( ___, observed_array, estimated_array, k0)  
[___]= sbioparamestim( ___, observed_array, estimated_array, k0,  
method)
```

### Arguments

<code>k</code>	Vector of estimated parameter values. For all optimization methods except 'fminsearch', the parameters are constrained to be greater than or equal to 0.
<code>result</code>	Structure with fields that provide information about the progress of optimization.
<code>modelObj</code>	SimBiology model object.
<code>tspan</code>	$n$ -by-1 vector representing the time span of the target data <code>xtarget</code> .
<code>xtarget</code>	$n$ -by- $m$ matrix, where $n$ is the number of time samples and $m$ is the number of states to match during the simulation. The number of rows in <code>xtarget</code> must equal the number of rows in <code>tspan</code> .

observed_array	<p>Either of the following:</p> <ul style="list-style-type: none"> <li>• Array of objects (species, compartment, or nonconstant parameter) in <code>modelObj</code>, whose values should be matched during the estimation process</li> <li>• Cell array of character vectors or string vector containing the names of objects (species, compartment, or nonconstant parameter) in <code>modelObj</code>, whose values should be matched during the estimation process</li> </ul> <hr/> <p><b>Note</b> If duplicate names exist for any species or parameters, ensure there are no ambiguities by specifying either an array of objects or a cell array of qualified names, such as <i>compartmentName.speciesName</i> or <i>reactionName.parameterName</i>. For example, for a species named <code>sp1</code> that is in a compartment named <code>comp2</code>, the qualified name is <code>comp2.sp1</code>.</p> <hr/> <p>The length of <code>observed_array</code> must equal the number of columns in <code>xtarget</code>. <code>sbioparamestim</code> assumes that the order of elements in <code>observed_array</code> is the same as the order of columns in <code>xtarget</code>.</p>
estimated_array	<p>Either of the following:</p> <ul style="list-style-type: none"> <li>• Array of objects (compartment, species, or parameter) in <code>modelObj</code> whose initial values should be estimated</li> <li>• Cell array of character vectors or string vector containing the names of objects (compartment, species, or parameter) in <code>modelObj</code> whose initial values should be estimated</li> </ul> <hr/> <p><b>Note</b> If duplicate names exist for any compartments, species, or parameters, ensure there are no ambiguities by specifying either an array of objects or a cell array of qualified names, such as <i>compartmentName.speciesName</i> or <i>reactionName.parameterName</i>. For example, for a parameter named <code>param1</code> scoped to a reaction named <code>reaction1</code>, the qualified name is <code>reaction1.param1</code>.</p>

<b>k0</b>	Numeric vector containing the initial values of compartments, species, or parameters to be estimated. The length of <b>k0</b> must equal that of <b>estimated_array</b> . If you do not specify <b>k0</b> , or specify an empty vector for <b>k0</b> , then <b>sbioparamestim</b> takes initial values for compartments, species, or parameters from <b>modelObj</b> , or, if there are active variants, <b>sbioparamestim</b> uses any initial values specified in the active variants. For details about variants, see <b>Variant</b> object.
-----------	---

method	<p>Optimization algorithm to use during the estimation process, specified by either of the following:</p> <ul style="list-style-type: none"> <li>• Character vector (or string) specifying one of the following functions: <ul style="list-style-type: none"> <li>• 'fminsearch'</li> <li>• 'lsqcurvefit'</li> <li>• 'lsqnonlin'</li> <li>• 'fmincon'</li> <li>• 'patternsearch'</li> <li>• 'patternsearch_hybrid'</li> <li>• 'ga'</li> <li>• 'ga_hybrid'</li> <li>• 'particleswarm'</li> <li>• 'particleswarm_hybrid'</li> </ul> </li> </ul> <p>For descriptions of how <code>sbioparamestim</code> uses the previous functions, see the Function Descriptions table.</p> <ul style="list-style-type: none"> <li>• Two-element cell array, with the first element being one of the previous functions, and the second element being an options structure or object. Use an appropriate options structure or object for each method listed next.</li> </ul> <table border="1" data-bbox="609 1112 1344 1529"> <thead> <tr> <th data-bbox="609 1112 977 1187">Method</th> <th data-bbox="977 1112 1344 1187">Options Structure or Object</th> </tr> </thead> <tbody> <tr> <td data-bbox="609 1187 977 1230">'fminsearch'</td> <td data-bbox="977 1187 1344 1230">optimset</td> </tr> <tr> <td data-bbox="609 1230 977 1326">'fmincon' 'lsqcurvefit'</td> <td data-bbox="977 1230 1344 1326">optimoptions</td> </tr> <tr> <td data-bbox="609 1326 977 1421">'lsqnonlin' 'particleswarm'</td> <td data-bbox="977 1326 1344 1421"></td> </tr> <tr> <td data-bbox="609 1421 977 1465">'particleswarm_hybrid'</td> <td data-bbox="977 1421 1344 1465"></td> </tr> <tr> <td data-bbox="609 1465 977 1529">'patternsearch'</td> <td data-bbox="977 1465 1344 1529"></td> </tr> </tbody> </table>	Method	Options Structure or Object	'fminsearch'	optimset	'fmincon' 'lsqcurvefit'	optimoptions	'lsqnonlin' 'particleswarm'		'particleswarm_hybrid'		'patternsearch'	
Method	Options Structure or Object												
'fminsearch'	optimset												
'fmincon' 'lsqcurvefit'	optimoptions												
'lsqnonlin' 'particleswarm'													
'particleswarm_hybrid'													
'patternsearch'													

<b>Method</b>	<b>Options Structure or Object</b>
'patternsearch_hybrid'  'ga'  'ga_hybrid'	

---

**Tip** Use a two-element cell array to provide your own options structure for the optimization algorithm.

---

If you have Parallel Computing Toolbox, you can enable parallel computing for faster data fitting by:

- 1** Opening a MATLAB worker pool:  
`parpool`
- 2** Setting the name-value pair argument 'UseParallel' to `true` in an options structure or object.



## Function Descriptions

Function	Description
fminsearch	<p>sbioparamestim uses the default options structure associated with fminsearch, except for:            Display = 'off'            TolFun = 1e-6* (Initial value of objective function)</p> <hr/> <p><b>Note</b> 'fminsearch' is an unconstrained optimization method, which can result in negative values for parameters.</p>
lsqcurvefit	<p>Requires Optimization Toolbox.</p> <p>sbioparamestim uses the default options structure associated with lsqcurvefit, except for:            Display = 'off'            FiniteDifferenceStepSize = value of the SolverOptions.RelativeTolerance property of the configuration set associated with modelObj, with a minimum of <math>\text{eps}^{(1/3)}</math>            FunctionTolerance = 1e-6* (Initial value of objective function)            TypicalX = 1e-6* (Initial values of components to be estimated)</p>
lsqnonlin	<p>Requires Optimization Toolbox.</p> <p>sbioparamestim uses the default options structure associated with lsqnonlin, except for:            Display = 'off'            FiniteDifferenceStepSize = value of the SolverOptions.RelativeTolerance property of the configuration set associated with <i>modelObj</i>, with a minimum of <math>\text{eps}^{(1/3)}</math>            FunctionTolerance = 1e-6* (Initial value of objective function)            TypicalX = 1e-6* (Initial values of components to be estimated)</p>

Function	Description
fmincon	<p>Requires Optimization Toolbox.</p> <p>sbioparamestim uses the default options structure associated with fmincon, except for:            Algorithm = 'interior-point'            Display = 'off'            FiniteDifferenceStepSize = value of the SolverOptions.RelativeTolerance property of the configuration set associated with <i>modelObj</i>, with a minimum of <math>\text{eps}^{(1/3)}</math>            FunctionTolerance = <math>1\text{e-}6*</math> (Initial value of objective function)            TypicalX = <math>1\text{e-}6*</math> (Initial values of components to be estimated)</p>
patternsearch	<p>Requires Global Optimization Toolbox.</p> <p>sbioparamestim uses the default options structure associated with patternsearch, except for:            Display = 'off'            FunctionTolerance = <math>1\text{e-}6*</math> (Initial value of objective function)            MeshTolerance = <math>1.0\text{e-}3</math>            AccelerateMesh = true</p>
patternsearch_hybrid	<p>Requires Global Optimization Toolbox.</p> <p>sbioparamestim calls the patternsearch function with the additional option SearchMethod = {@searchlhs,10,15}. This option adds an additional search step that uses Latin hypercube sampling.</p> <p>The sbioparamestim function uses the default options structure associated with patternsearch, except for:            Display = 'off'            FunctionTolerance = <math>1\text{e-}6*</math> (Initial value of objective function)            MeshTolerance = <math>1.0\text{e-}3</math>            AccelerateMesh = true            SearchMethod = {@searchlhs,10,15}</p>

Function	Description
ga	<p>Requires Global Optimization Toolbox.</p> <p>sbioparamestim uses the default options structure associated with <code>ga</code>, except for:</p> <pre>Display = 'off' FunctionTolerance = 1e-6* (Initial value of objective function) PopulationSize = 10 Generations = 30 MutationFcn = @mutationadaptfeasible</pre>
ga_hybrid	<p>Requires Global Optimization Toolbox.</p> <p>sbioparamestim calls the <code>ga</code> function with the additional option <code>HybridFcn = {@fmincon, fminopt}</code>, where <code>fminopt</code> is the same set of default options <code>sbioparamestim</code> uses for <code>fmincon</code>. This option causes an additional gradient-based minimization after the genetic algorithm step ends.</p> <p>The <code>sbioparamestim</code> function uses the default options structure associated with <code>ga</code>, except for:</p> <pre>Display = 'off' FunctionTolerance = 1e-6* (Initial value of objective function) PopulationSize = 10 Generations = 30 MutationFcn = @mutationadaptfeasible HybridFcn = {@fmincon, structure of name/value pairs for fmincon}</pre>
particleswarm	<p>Requires Global Optimization Toolbox.</p> <p>sbioparamestim uses the following default options for <code>particleswarm</code>, except for:</p> <pre>Display          = 'off'; FunctionTolerance = 1e-6*[Initial objective function value] SwarmSize        = 10; MaxIter          = 30;</pre>

Function	Description
particleswarm_hybrid	<p>Requires Global Optimization Toolbox.</p> <p>sbioparamestim calls the particleswarm function with the additional option HybridFcn = {@objFcn, options}. The objective function, objFcn, is one of these supported functions: patternsearch, fminsearch, fminunc, or fmincon. options is a structure of options for these functions and their values.</p> <pre> Display          = 'off'; FunctionTolerance = 1e-6*[Initial objective function value]; SwarmSize        = 10; MaxIter          = 30; HybridFcn        = {@fmincon, [Fmincon Options, described above]}                     </pre>

---

**Note** sbioparamestim does not support setting the Vectorized option to 'on' in algorithms that support this option.

---

## Description

[k, result]= sbioparamestim(modelObj, tspan, xtarget, observed\_array, estimated\_array) estimates the initial values of compartments, species, and parameters of modelObj, a SimBiology model object, specified in estimated\_array, so as to match the values of species and nonconstant parameters given by observed\_array with the target state, xtarget, whose time variation is given by the time span tspan. If you have Optimization Toolbox installed, sbioparamestim uses the lsqnonlin function as the default method for the parameter estimation. If you do not have Optimization Toolbox installed, sbioparamestim uses the MATLAB function fminsearch as the default method for the parameter estimation.

[\_\_\_]= sbioparamestim(\_\_\_ , observed\_array, estimated\_array, k0) specifies the initial values of compartments, species, and parameters listed in estimated\_array.

[\_\_\_]= sbioparamestim(\_\_\_ , observed\_array, estimated\_array, k0, method) specifies the optimization method to use.

## Examples

Given a model and some target data, estimate all of its parameters without explicitly specifying any initial values:

- 1 Load a model from the project, `gprotein_norules.sbproj`. The project contains two models, one for the wild-type strain (stored in variable `m1`), and one for the mutant strain (stored in variable `m2`). Load the G protein model for the wild-type strain.

```
sbioloadproject gprotein_norules m1;
```

- 2 Store the target data in a variable:

```
Gt = 10000;
tspan = [0 10 30 60 110 210 300 450 600]';
Ga_frac = [0 0.35 0.4 0.36 0.39 0.33 0.24 0.17 0.2]';
xtarget = Ga_frac * Gt;
```

- 3 Store all model parameters in an array:

```
p_array = sbiiselect(m1, 'Type', 'parameter');
```

- 4 Store the species that should match target:

```
Ga = sbiiselect(m1, 'Type', 'species', 'Name', 'Ga');
% In this example only one species is selected.
% To match more than one targeted species data
% replace with selected species array.
```

- 5 Estimate the parameters:

```
[k, result] = sbioparamestim(m1, tspan, xtarget, Ga, p_array)
```

```
k =
```

```
0.0100
0.0000
0.0004
4.0000
0.0040
1.0000
0.0000
0.1100
```

```
result =
```

```
        fval: 1.4193e+06
        residual: [9x1 double]
        exitflag: 2
        iterations: 2
        funccount: 27
        algorithm: 'trust-region-reflective'
        message: [1x413 char]
```

Estimate parameters specified in `p_array` for species Ga using different algorithms.

```
[k1,r1] = sbioparamestim(m1,tspan,xtarget,Ga,p_array, ...
    {},'fmincon');
[k2,r2] = sbioparamestim(m1,tspan,xtarget,Ga,p_array, ...
    {},'patternsearch');
[k3,r3] = sbioparamestim(m1,tspan,xtarget,Ga,p_array, ...
    {},'ga');
[k4,r4] = sbioparamestim(m1,tspan,xtarget,Ga,p_array, ...
    {},'particleswarm');
```

Estimate parameters specified in `p_array` for species Ga, and change default optimization options to use user-specified options.

```
myopt1 = optimoptions('Display','iter');
[k1,r1] = sbioparamestim(m1,tspan,xtarget, ...
    Ga,p_array,{},{'fmincon',myopt1});

myopt2 = optimoptions('MeshTolerance',1.0e-4);
[k2,r2] = sbioparamestim(m1,tspan,xtarget, ...
    Ga,p_array,{},{'patternsearch',myopt2});

myopt3 = optimoptions('PopulationSize',25, 'Generations', 10);
[k3,r3] = sbioparamestim(m1,tspan,xtarget, ...
    Ga,p_array,{},{'ga',myopt3});

myopt4 = optimoptions('particleswarm','Display','iter');
[k4,r4] = sbioparamestim(m1,tspan,xtarget,Ga,p_array,{},{'particleswarm',myopt4});
```

## Algorithms

`sbioparamestim` estimates parameters by attempting to minimize the discrepancy between simulation results and the data to fit. The minimization uses one of these optimization algorithms: `fminsearch` (from MATLAB); `lsqcurvefit`, `lsqnonlinfit`,

or `fmincon` (from Optimization Toolbox); or `patternsearch` or `ga` (from Global Optimization Toolbox). All optimization methods require an objective function as an input. This objective function takes as input a vector of parameter values and returns an estimate of the discrepancy between simulation and data. When using `lsqcurvefit` or `lsqnonlinfit` as the optimization method, this objective function returns a vector of the residuals. For other optimization methods, the objective function returns the 2-norm of the residuals.

## References

- [1] Yi, T.-M., Kitano, H., and Simon, M.I. (2003) A quantitative characterization of the yeast heterotrimeric G protein cycle. *PNAS* *100*, 10764–10769.

## See Also

`optimset` | `sbiomodel`

**Introduced in R2006a**

## sbiopredictionci

Compute confidence intervals for model predictions (requires Statistics and Machine Learning Toolbox)

### Syntax

```
ci = sbiopredictionci(fitResults)
ci = sbiopredictionci(fitResults,Name,Value)
```

### Description

`ci = sbiopredictionci(fitResults)` computes 95% confidence intervals for the model simulation results from `fitResults`, an `NLINResults` object or `OptimResults` object returned by `sbiofit`. `ci` is a `PredictionConfidenceInterval` object that contains the computed confidence interval data.

`ci = sbiopredictionci(fitResults,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

### Examples

#### Compute Confidence Intervals for Estimated PK Parameters and Model Predictions

##### Load Data

Load the sample data to fit. The data is stored as a table with variables *ID*, *Time*, *CentralConc*, and *PeripheralConc*. This synthetic data represents the time course of plasma concentrations measured at eight different time points for both central and peripheral compartments after an infusion dose for three individuals.

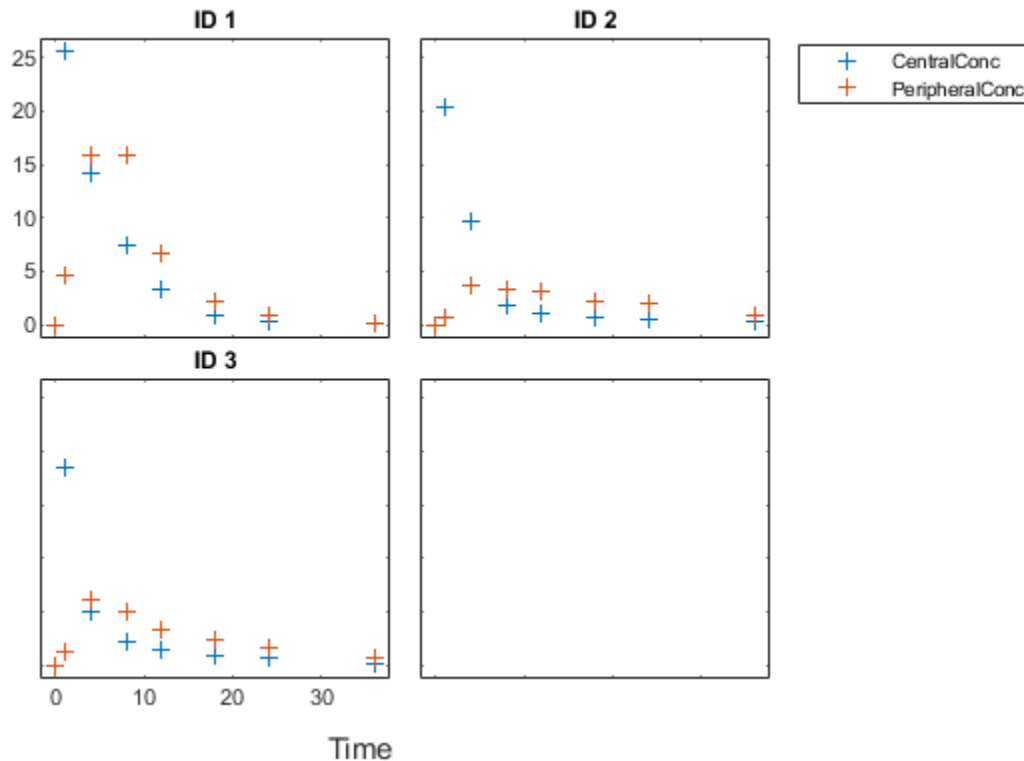
```
clear all
load data10_32R.mat
```



```

gData = groupedData(data);
gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};
sbiotrellis(gData,'ID','Time',{ 'CentralConc','PeripheralConc'}, 'Marker','+',...
            'LineStyle','none');

```



## Create Model

Create a two-compartment model.

```

pkmd = PKModelDesign;
pkcl = addCompartment(pkmd,'Central');
pkcl.DosingType = 'Infusion';
pkcl.EliminationType = 'linear-clearance';
pkcl.HasResponseVariable = true;

```

```
pkc2          = addCompartment(pkmd, 'Peripheral');
model        = construct(pkmd);
configset    = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

### Define Dosing

Define the infusion dose.

```
dose          = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime = 0;
dose.Amount   = 100;
dose.Rate     = 50;
dose.AmountUnits = 'milligram';
dose.TimeUnits = 'hour';
dose.RateUnits = 'milligram/hour';
```

### Define Parameters

Define the parameters to estimate. Set the parameter bounds for each parameter. In addition to these explicit bounds, the parameter transformations (such as log, logit, or probit) impose implicit bounds.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam = estimatedInfo(paramsToEstimate, ...
                               'InitialValue', [1 1 1 1], ...
                               'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

### Fit Model

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

Perform a pooled fit, that is, one set of estimated parameters for all patients.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true);
```

### Compute Confidence Intervals for Estimated Parameters

Compute 95% confidence intervals for each estimated parameter in the unpooled fit.

```
ciParamUnpooled = sbioparameterci(unpooledFit);
```

## Display Results

Display the confidence intervals in a table format. For details about the meaning of each estimation status, see “Parameter Confidence Interval Estimation Status” on page 2-374.

```
ci2table(ciParamUnpooled)
```

```
ans =
```

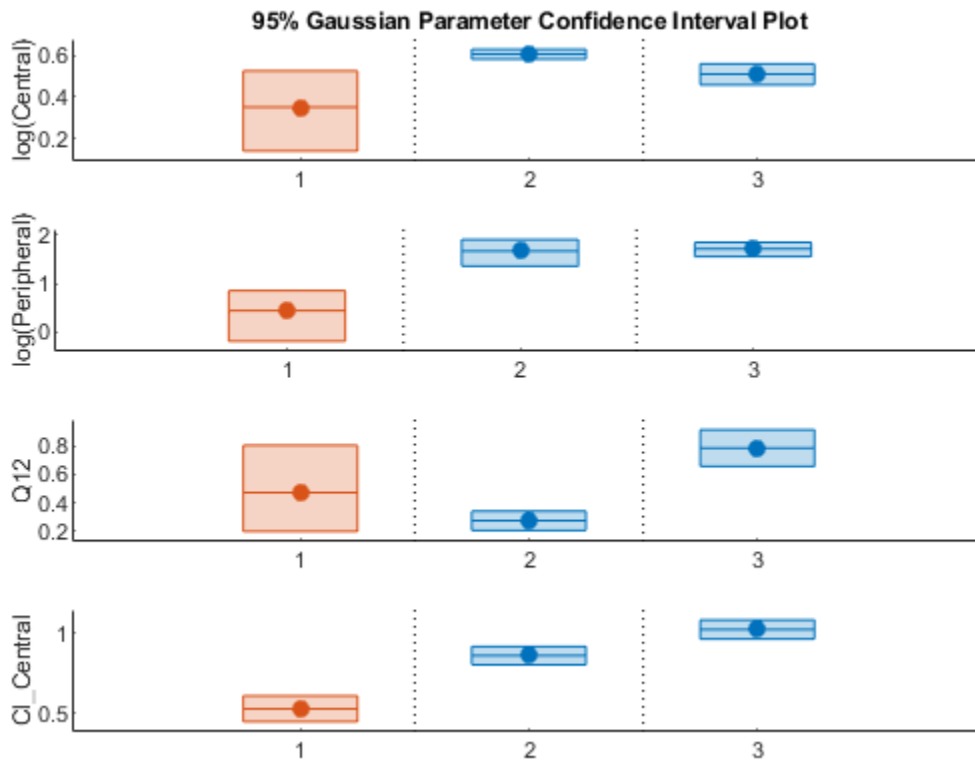
```
12x7 table
```

Group	Name	Estimate	ConfidenceInterval		Type	Alpha
1	'Central'	1.422	1.1533	1.6906	Gaussian	0.05
1	'Peripheral'	1.5629	0.83143	2.3551	Gaussian	0.05
1	'Q12'	0.47159	0.20093	0.80247	Gaussian	0.05
1	'Cl_Central'	0.52898	0.44842	0.60955	Gaussian	0.05
2	'Central'	1.8322	1.7893	1.8751	Gaussian	0.05
2	'Peripheral'	5.3368	3.9133	6.7602	Gaussian	0.05
2	'Q12'	0.27641	0.2093	0.34351	Gaussian	0.05
2	'Cl_Central'	0.86034	0.80313	0.91755	Gaussian	0.05
3	'Central'	1.6657	1.5818	1.7497	Gaussian	0.05
3	'Peripheral'	5.5632	4.7557	6.3708	Gaussian	0.05
3	'Q12'	0.78361	0.65581	0.91142	Gaussian	0.05
3	'Cl_Central'	1.0233	0.96375	1.0828	Gaussian	0.05

Plot the confidence intervals. If the estimation status of a confidence interval is **success**, it is plotted in blue (the first default color). Otherwise, it is plotted in red (the second default color), which indicates that further investigation into the fitted parameters may be required. If the confidence interval is **not estimable**, then the function plots a red line with a centered cross. If there are any transformed parameters with estimated values 0 (for the log transform) and 1 or 0 (for the probit or logit transform), then no confidence intervals are plotted for those parameter estimates. To see the color order, type `get(groot, 'defaultAxesColorOrder')`.

Groups are displayed from left to right in the same order that they appear in the `GroupNames` property of the object, which is used to label the x-axis. The y-labels are the transformed parameter names.

```
plot(ciParamUnpooled)
```



Compute the confidence intervals for the pooled fit.

```
ciParamPooled = sbioparameterci(pooledFit);
```

Display the confidence intervals.

```
ci2table(ciParamPooled)
```

ans =

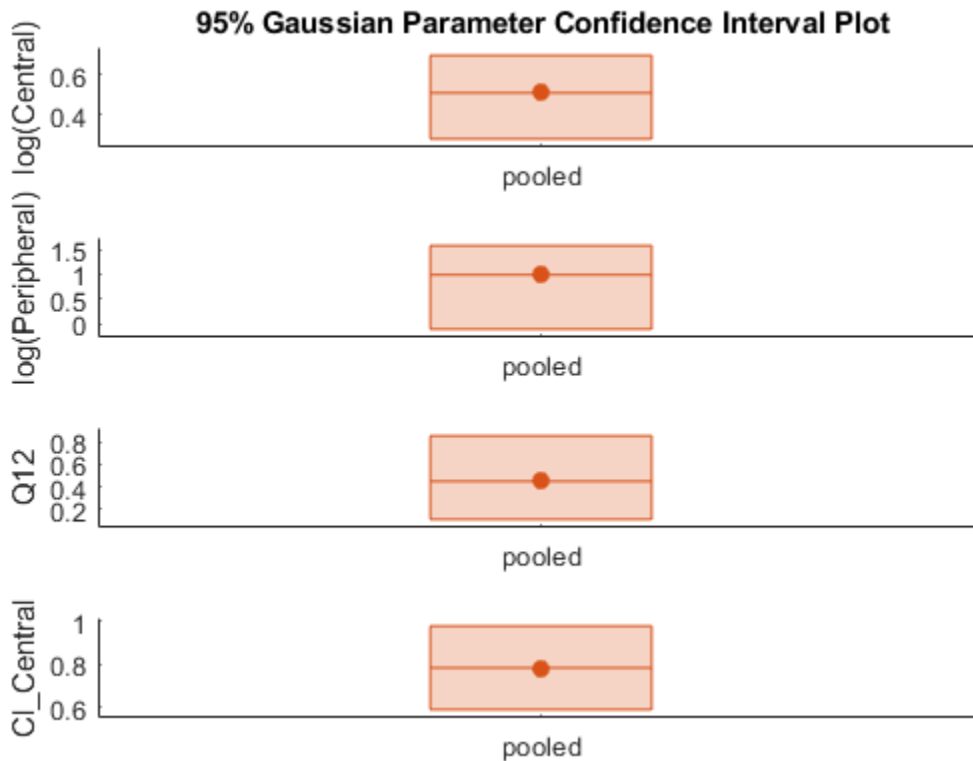
4x7 table

Group	Name	Estimate	ConfidenceInterval	Type	Alpha
-------	------	----------	--------------------	------	-------

pooled	'Central'	1.6626	1.3287	1.9965	Gaussian	0.05	es
pooled	'Peripheral'	2.687	0.89848	4.8323	Gaussian	0.05	co
pooled	'Q12'	0.44956	0.11445	0.85152	Gaussian	0.05	co
pooled	'Cl_Central'	0.78493	0.59222	0.97764	Gaussian	0.05	es

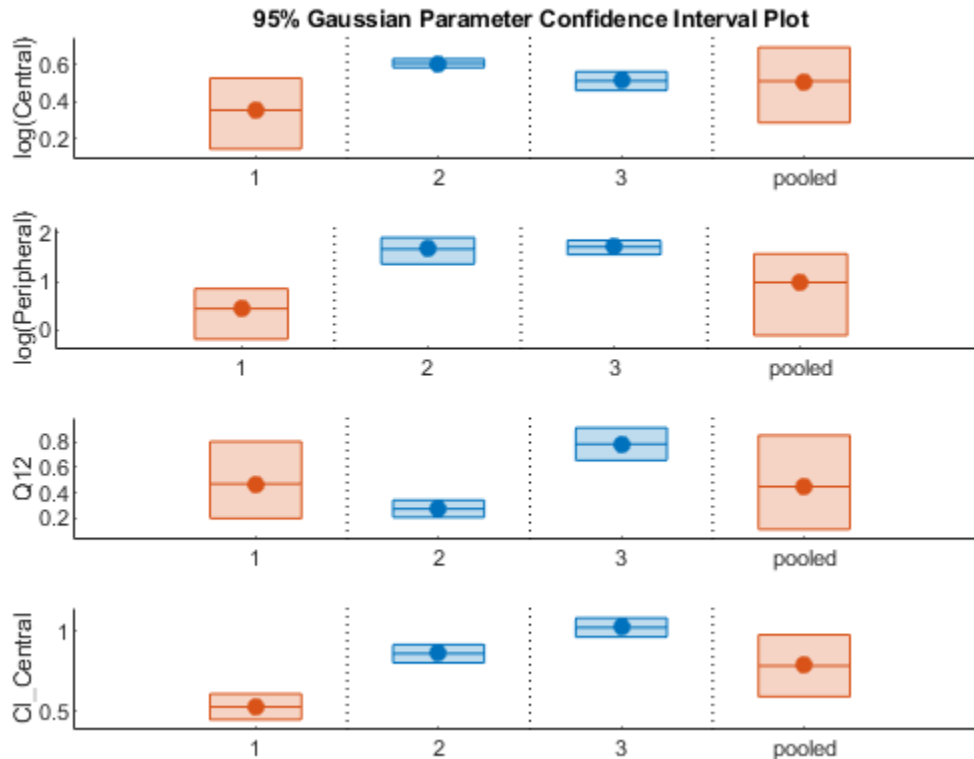
Plot the confidence intervals. The group name is labeled as "pooled" to indicate such fit.

```
plot(ciParamPooled)
```



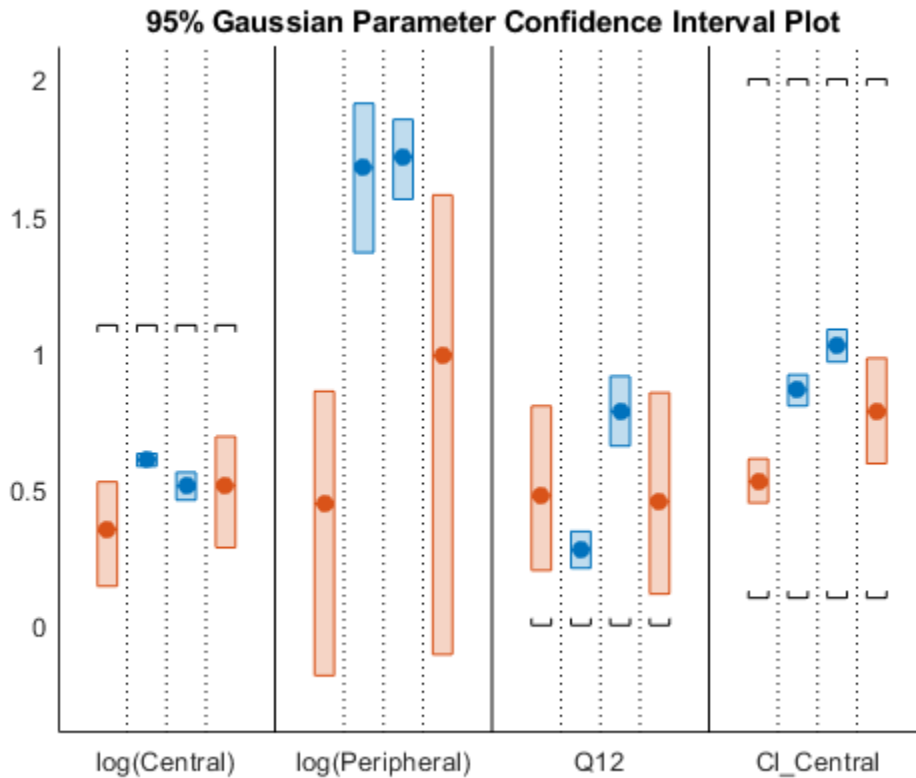
Plot all the confidence interval results together. By default, the confidence interval for each parameter estimate is plotted on a separate axes. Vertical lines group confidence intervals of parameter estimates that were computed in a common fit.

```
ciAll = [ciParamUnpooled;ciParamPooled];
plot(ciAll)
```



You can also plot all confidence intervals in one axes grouped by parameter estimates using the 'Grouped' layout.

```
plot(ciAll, 'Layout', 'Grouped')
```



In this layout, you can point to the center marker of each confidence interval to see the group name. Each estimated parameter is separated by a vertical black line. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets. Note the different scales on the y-axis due to parameter transformations. For instance, the y-axis of Q12 is in the linear scale, but that of Central is in the log scale due to its log transform.

### Compute Confidence Intervals for Model Predictions

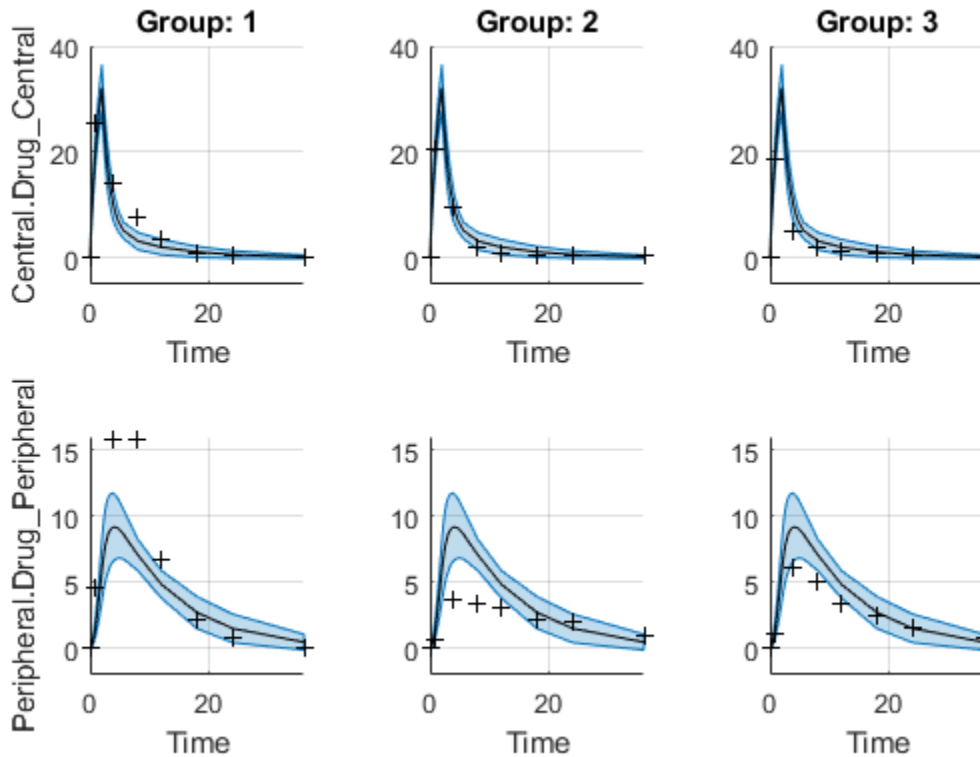
Calculate 95% confidence intervals for the model predictions, that is, simulation results using the estimated parameters.

```
% For the pooled fit
ciPredPooled = sbiopredictionci(pooledFit);
% For the unpooled fit
ciPredUnpooled = sbiopredictionci(unpooledFit);
```

### Plot Confidence Intervals for Model Predictions

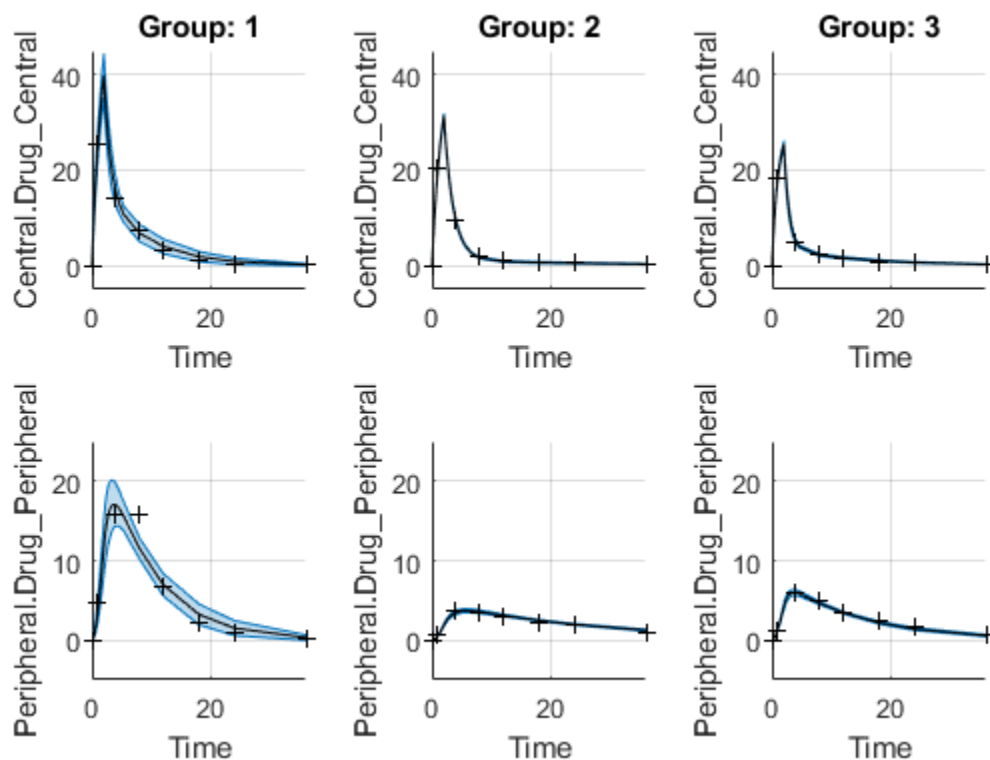
The confidence interval for each group is plotted in a separate column, and each response is plotted in a separate row. Confidence intervals limited by the bounds are plotted in red. Confidence intervals not limited by the bounds are plotted in blue.

```
plot(ciPredPooled)
```



```
plot(ciPredUnpooled)
```





## Input Arguments

### **fitResults** – Parameter estimation results from `sbiofit`

NLINResults object | OptimResults object | vector

Parameter estimation results from `sbiofit`, specified as an `NLINResults` object, `OptimResults` object, or a vector of objects for unpooled fits that were returned from the same `sbiofit` call.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Alpha', 0.01, 'Type', 'bootstrap'` specifies to compute a 99% confidence interval using the bootstrap method.

### **Alpha — Confidence level**

0.05 (default) | positive scalar

Confidence level,  $(1 - \textit{Alpha}) * 100\%$ , specified as the comma-separated pair consisting of `'Alpha'` and a positive scalar between 0 and 1. The default value is 0.05, meaning a 95% confidence interval is computed.

Example: `'Alpha', 0.01`

### **Type — Confidence interval type**

`'gaussian'` (default) | `'bootstrap'`

Confidence interval type, specified as the comma-separated pair consisting of `'Type'` and a character vector. The valid choices are:

- `'gaussian'` - Use the Gaussian approximation on page 1-238 of the distribution of the linearized model responses around the parameter estimates.
- `'bootstrap'` - Compute confidence intervals using the bootstrap method on page 1-239.

Example: `'Type', 'bootstrap'`

### **NumSamples — Number of samples for bootstrapping**

1000 (default) | positive integer

Number of samples for bootstrapping, specified as the comma-separated pair consisting of `'NumSamples'` and a positive integer. This number defines the number of fits that are performed during the confidence interval computation to generate bootstrap samples. The smaller the number is, the faster the computation of the confidence intervals becomes, at the cost of decreased accuracy.

Example: `'NumSamples', 500`

**Display** – Level of display returned to the command line`'off' (default) | 'none' | 'final'`

Level of display returned to the command line, specified as the comma-separated pair consisting of `'Display'` and a character vector. `'off'` (default) or `'none'` displays no output. `'final'` displays a message when the computation finishes.

Example: `'Display','final'`

**UseParallel** – Logical flag to compute confidence intervals in parallel`true | false`

Logical flag to compute confidence intervals in parallel, specified as the comma-separated pair consisting of `'UseParallel'` and `true` or `false`. By default, the parallel options in the original fit are used. If this argument is set to `true` and Parallel Computing Toolbox is available, the parallel options in the original fit are ignored, and confidence intervals are computed in parallel.

For the Gaussian confidence intervals:

- If the input `fitResults` is a vector of results objects, then the computation of confidence intervals for each object is performed in parallel. The Gaussian confidence intervals are quick to compute. So, it might be more beneficial to parallelize the original fit (`sbiofit`) and not set `UseParallel` to `true` for `sbiopredictionci`.

For the Bootstrap confidence intervals:

- The function forwards the `UseParallel` flag to `bootci`. There is no parallelization over the input vector of results objects.

---

**Note** If you have a global stream for random number generation with a number of substreams to compute in parallel in a reproducible fashion, `sbiopredictionci` first checks to see if the number of workers is same as the number of substreams. If so, the function sets `UseSubstreams` to `true` in the `statset` option and passes to `bootci`. Otherwise, the substreams are ignored by default.

---

Example: `'UseParallel',true`

## Output Arguments

### **ci** — Confidence interval results

PredictionConfidenceInterval object

Confidence interval results, returned as a PredictionConfidenceInterval object. For an unpooled fit, ci can be a vector of PredictionConfidenceInterval objects.

## Definitions

### **Gaussian Confidence Interval Calculation for Model Predictions**

The model is linearized around the parameter estimates  $P_{est}$  that are obtained from the fit results returned by sbiofit. The CovarianceMatrix is transformed using the linearized model. In addition, implicit parameter bounds (log, probit, or logit parameter transforms specified in the original fit) and explicit parameter bounds (if specified in the original fit) are also mapped through the linearized model.

To linearize the model, sbiopredictionci first checks to see if the sensitivity analysis feature is turned on in the original fit. If the feature is on, the function uses the Jacobian computed via the complex step differentiation. If the feature is off, the Jacobian is computed using finite differencing. Finite differencing can be inaccurate, and consider turning on the sensitivity analysis on page 1-0 feature when you run sbiofit.

The function uses the transformed CovarianceMatrix and computes the Gaussian confidence intervals for each estimated model response at every time step.

In cases where the confidence interval is constrained by the parameter bounds defined in the original fit, the confidence interval bounds are adjusted according to the approach described by Wu, H. and Neale, M. [1].

- For each model response, the function first decides whether the confidence interval is unbounded. If so, the estimation status of the corresponding model response is set to **not estimable**.
- Otherwise, if the confidence interval for a response is constrained by a parameter bound defined in the original fit, the function sets its status to **constrained**.

Parameter transformations (such as `log`, `probit`, or `logit`) impose implicit bounds on the estimated parameters, for example, positivity constraints. Such bounds can lead to the overestimation of confidence, that is, the confidence interval can be smaller than expected.

- If no confidence interval has the estimation status `not estimable` or `constrained`, then the function sets the estimation statuses of all model responses to `success`. Otherwise, the estimation statuses of remaining model responses are set to `estimable`.

## Bootstrap Confidence Interval Calculation

The `bootci` function from Statistics and Machine Learning Toolbox is used to compute the bootstrap confidence intervals. The first input `nboot` is the number of samples (`NumSamples`), and the second input `bootfun` is a function that performs these actions.

- Resample the data (independently within each group, if multiple groups are available).
- Run a parameter fit with the resampled data.
- Simulate the model using the estimated parameters to get model responses.
- Return model responses.

The estimation status is always set to `estimable` since the function cannot determine if the confidence intervals are constrained by the bounds on the parameter estimates.

## References

- [1] Wu, H., and M.C. Neale. "Adjusted Confidence Intervals for a Bounded Parameter." *Behavior Genetics*. 42 (6), 2012, pp. 886-898.

## Extended Capabilities

### Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true`.

For more information, see the 'UseParallel' name-value pair argument.

## **See Also**

ParameterConfidenceInterval | PredictionConfidenceInterval | sbiofit |  
sbioparameterci

**Introduced in R2017b**

# sbioplot

Plot simulation results in one figure

## Syntax

```
sbioplot(sd)
sbioplot(sd,fcnHandle,xArgs,yArgs,Name,Value)
```

## Description

`sbioplot(sd)` plots each simulation run from `sd`, a `SimData` object or array of objects, in the same figure. The plot is a time plot of each state in `sd`. The figure also shows a hierarchical display of all the runs as different nodes in a tree, and you can select which run to display.

`sbioplot(sd,fcnHandle,xArgs,yArgs,Name,Value)` plots simulation results by calling the function handle `fcnHandle` with inputs `sd`, `xArgs`, and `yArgs`, and uses additional options specified by one or more name-value pair arguments. For example, you can specify the x-label and y-label of the plot. `xArgs` and `yArgs` must be cell arrays or string vectors of the names of the states to plot.

## Examples

### Plot Selected States from Simulation Data

Plot the prey versus predator data from the stochastically simulated lotka model by using a custom function (`plotXY`).

Load the model. Set the solver type to SSA to perform stochastic simulations, and set the stop time to 3.

```
sbioloadproject lotka;
cs                = getconfigset(m1);
cs.SolverType    = 'SSA';
```

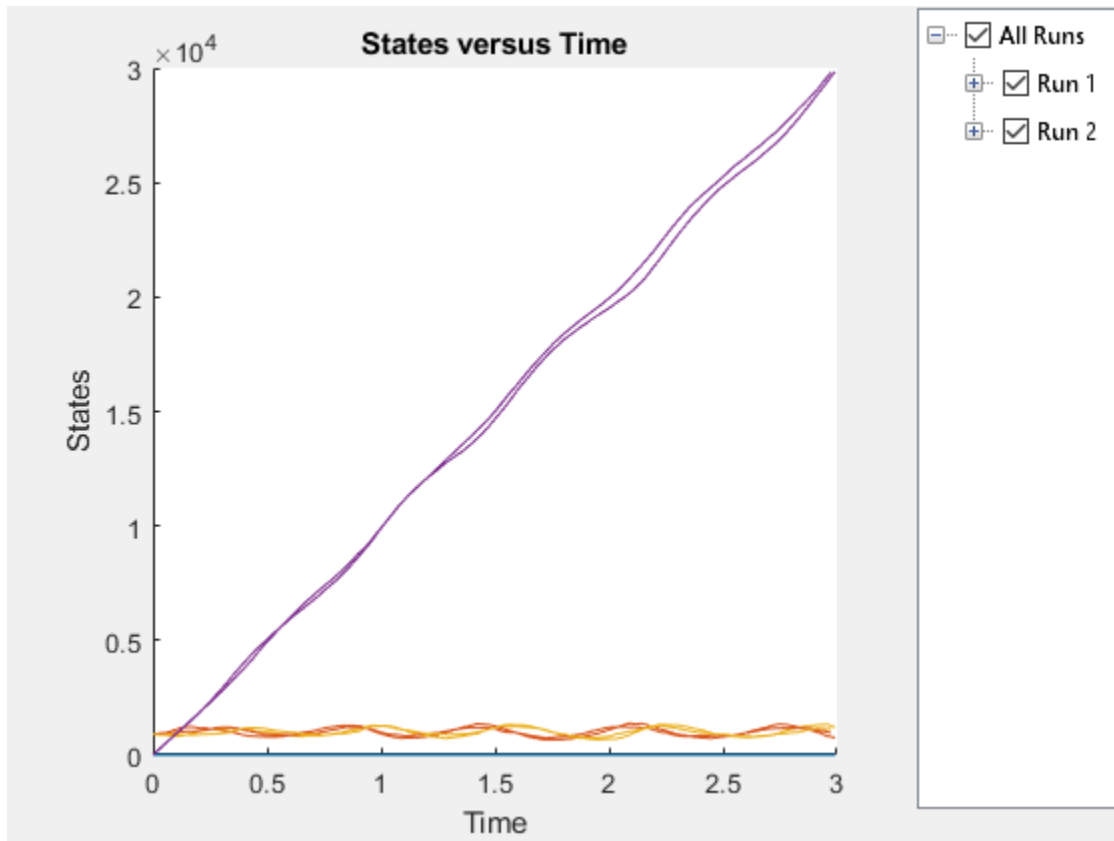
```
cs.StopTime = 3;
rng('default') % For reproducibility
```

Set the number of runs and use `sbioensemblerun` for simulation.

```
numRuns = 2;
sd = sbioensemblerun(m1,numRuns);
```

Plot the simulation data. By default, `sbioplot` shows the time plot of each species for each run. You can expand each run by clicking the + icon, and you can select individual time plots.

```
sbioplot(sd);
```

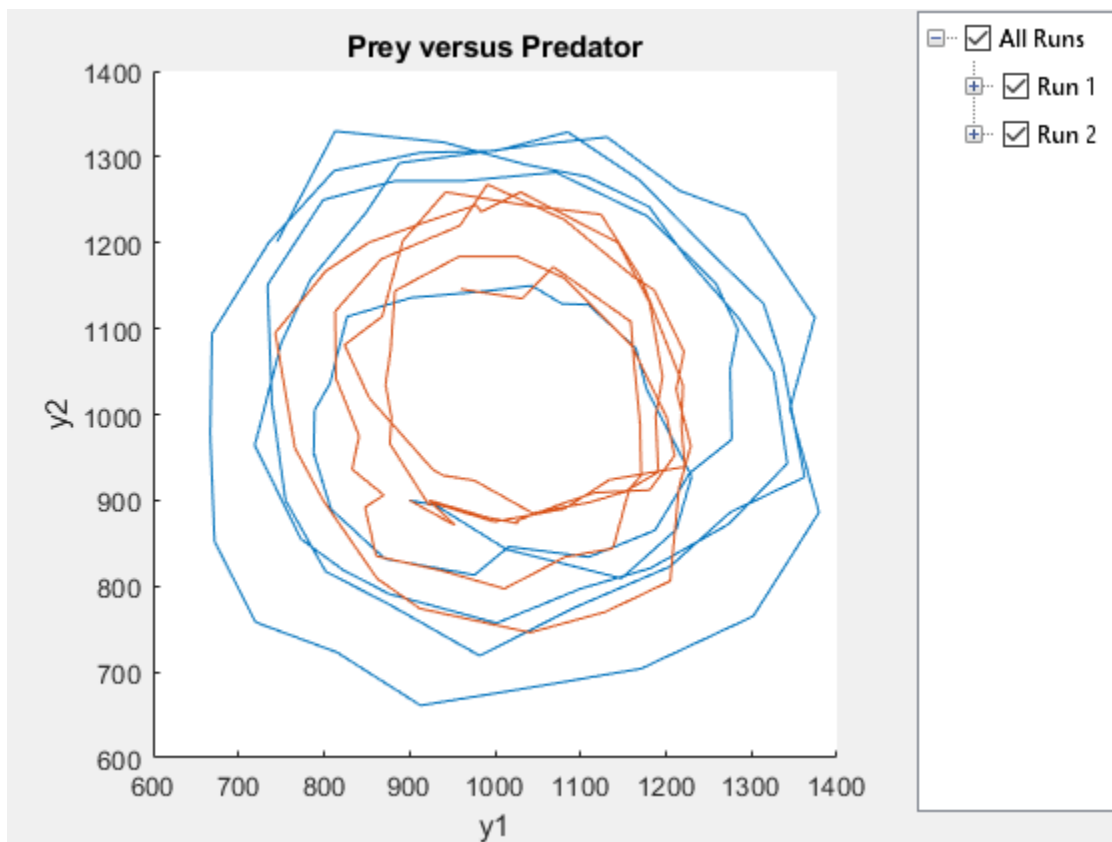




Plot selected states against each other; in this case, plot the prey population versus the predator population. Use the function `plotXY` (shown at the end of this example) to plot the simulated `y1` (prey) data versus the `y2` (predator) data. Specify the function as a function handle.

If you use the live script file for this example, the `plotXY` function is already included at the end of the file. Otherwise, you must define the `plotXY` function at the end of your `.m` or `.mlx` file or add it as a file on the MATLAB path.

```
sbioplot(sd,@plotXY,{'y1'},{'y2'},'xlabel','y1','ylabel','y2','title','Prey versus Predator')
```



### Define `plotXY` Function

`sbioplot` accepts a function handle for a function with the signature:

```
function [handles,names] = functionName(sd,xArgs,yArgs).
```

The `plotXY` function plots two selected states against each other. The first input `sd` is the simulation data (SimBiology `SimData` object or vector of objects). In this particular example, `xArgs` is a cell array containing the name of the species to be plotted on the x-axis, and `yArgs` is a cell array containing the name of the second species to be plotted on the y-axis. However, you can use the inputs `xArgs` and `yArgs` in any way in *your* custom plotting function. The function returns `handles`, an array of function handles to the line plots, and `names`, a cell array of character vectors shown on the nodes that are children of a **Run** node in a hierarchical display.

```
function [handles,names] = plotXY(sd,xArgs,yArgs)
```

```
% Select simulation data for each state from each run.
```

```
xData1 = selectbyname(sd(1),xArgs);
```

```
xData2 = selectbyname(sd(2),xArgs);
```

```
yData1 = selectbyname(sd(1),yArgs);
```

```
yData2 = selectbyname(sd(2),yArgs);
```

```
% Plot the species against each other.
```

```
fH1 = plot(xData1.Data,yData1.Data);
```

```
fH2 = plot(xData2.Data,yData2.Data);
```

```
% The first output, handles, is a two-dimensional array of handles of the line plots. I
```

```
% where M is the number of line plots for each run and N is the number of runs.
```

```
handles = [fH1,fH2];
```

```
% The second output, names, must be a one-dimensional cell array of character vectors.
```

```
% Its length must be equal to the number of rows in handles, and the texts are displaye
```

```
% nodes that are children of a Run node.
```

```
names = {'y1 vs y2'};
```

```
end
```

## Input Arguments

### **sd** — Simulation results

`SimData` object

Simulation results, specified as a `SimData` object or vector of `SimData` objects.

This argument corresponds to the first input of the function referenced by `fcnHandle`.

Example: `simdata`

### **fcnHandle** — Function to generate line plots

function handle

Function to generate line plots, specified as a function handle. For an example of a custom function to plot selected species from simulation data, see Plot Selected States from Simulation Data on page 1-241.

The function must have the signature:

```
function [handles, names] = functionName(sd, xArgs, yArgs).
```

The inputs `sd`, `xArgs`, and `yArgs` are the same inputs that you pass in when you call `sbioplot`.

The first output `handles` is a two-dimensional array of handles of the line plots generated by the function. Its size must be  $P$ -by- $R$ , where  $P$  is the number of line plots, and  $R$  is the number of runs.

The second output `names` is a one-dimensional cell array of character vectors containing the names to be displayed on the nodes that are children of a **Run** node in a hierarchical display. The length of `names` must be equal to the number of rows in `handles`.

Example: `@plotXY`

Data Types: `function_handle`

### **xArgs** — State names

string vector | cell array of character vectors

State names to plot, specified as a string vector or cell array of character vectors. For instance, you can use `xArgs` to represent the states to be plotted on the x-axis of your custom plot.

This argument corresponds to the second input of the function referenced by `fcnHandle`.

Example: `{'y1'}`

Data Types: `cell`

### **yArgs** — State names

string vector | cell array of character vectors

State names to plot, specified as a string vector or cell array of character vectors. For instance, you can use `yArgs` to represent the states to be plotted on the y-axis of your custom plot.

This argument corresponds to the third input of the function referenced by `fcnHandle`.

Example: `{'y2', 'z'}`

Data Types: `cell`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'title', 'Species X versus Species Y'` specifies the axes title of the plot.

### **title — Axes title**

character vector | string

Axes title, specified as the comma-separated pair consisting of `'title'` and character vector or string.

Example: `'title', 'Prey versus Predator'`

Data Types: `char` | `string`

### **xlabel — Label for x-axis**

character vector | string

Label for the x-axis of the plot, specified as the comma-separated pair consisting of `'xlabel'` and a character vector or string.

Example: `'xlabel', 'y1'`

Data Types: `char` | `string`

### **ylabel — Label for y-axis**

character vector | string

Label for the y-axis of the plot, specified as the comma-separated pair consisting of `'ylabel'` and a character vector or string.

Example: 'ylabel', 'y2'

Data Types: char | string

## **See Also**

[sbiosubplot](#) | [SimData](#)

**Introduced in R2008a**

## **sbioremovefromlibrary**

Remove kinetic law, unit, or unit prefix from library

### **Syntax**

```
sbioremovefromlibrary (Obj)  
sbioremovefromlibrary ('Type', 'Name')
```

### **Description**

`sbioremovefromlibrary (Obj)` removes the kinetic law definition, unit, or unit prefix object (Obj) from the user-defined library. The removed component will no longer be available automatically in future MATLAB sessions.

`sbioremovefromlibrary` does not remove a kinetic law definition that is being used in a model.

You can use a built-in or user-defined kinetic law definition when you construct a kinetic law object with the method `addkineticlaw`.

`sbioremovefromlibrary ('Type', 'Name')` removes the object of type 'Type' with name 'Name' from the corresponding user-defined library. Type can be 'kineticlaw', 'unit' or 'unitprefix'.

To get a component of the built-in and user-defined libraries, use the commands `get(sbioroot, 'BuiltInLibrary')` and `get(sbioroot, 'UserDefinedLibrary')`.

To create a kinetic law definition, unit, or unit prefix, use `sbioabstractkineticlaw`, `sbiounit`, or `sbiounitprefix` respectively.

To add a kinetic law definition, unit, or unit prefix to the user-defined library, use the function `sbioaddtolibrary`.

## Examples

This example shows how to remove a kinetic law definition from the user-defined library.

- 1 Create a kinetic law definition.

```
abstkineticlawObj = sbioabstractkineticlaw('mylaw1', '(k1*s)/(k2+k1+s)');
```

- 2 Add the new kinetic law definition to the user-defined library.

```
sbioaddtolibrary(abstkineticlawObj);
```

`sbioaddtolibrary` adds the kinetic law definition to the user-defined library. You can verify this using `sbiowhos`.

```
sbiowhos -kineticlaw -userdefined
```

```
SimBiology Abstract Kinetic Law Array
```

Index:	Library:	Name:	Expression:
1	UserDefined	mylaw1	(k1*s)/(k2+k1+s)

- 3 Remove the kinetic law definition.

```
sbioremovefromlibrary('kineticlaw', 'mylaw1');
```

## See Also

[sbioabstractkineticlaw](#) | [sbioaddtolibrary](#) | [sbiounit](#) | [sbiounitprefix](#)

**Introduced in R2006a**

## **sbioreset**

Delete all model objects

### **Syntax**

```
sbioreset
```

### **Description**

`sbioreset` deletes all SimBiology model objects at the root level. You cannot use a SimBiology model object after it is deleted.

---

**Tip** To remove a SimBiology model object from the MATLAB workspace, without deleting it from the root level, use the `clear` function.

---

---

**Note** If the SimBiology desktop is open, calling `sbioreset` at the command line deletes all model objects that are open in the desktop.

---

The SimBiology root object contains a list of SimBiology model objects, available units, unit prefixes, and kinetic law objects. A SimBiology model object has its `Parent` property set to the SimBiology root object.

To add a kinetic law definition to the SimBiology root user-defined library, use the `sbioaddtolibrary` function. To add a unit to the SimBiology user-defined library on the root, use `sbiounit` followed by `sbioaddtolibrary`. To add a unit prefix to the SimBiology user-defined library on the root, use `sbiounitprefix` followed by `sbioaddtolibrary`.

### **Examples**

This example shows the difference between `sbioreset` and `clear all`.



- 1 Import a model into the workspace.

```
modelObj = sbmlimport('oscillator');
```

Note that the workspace contains `modelObj` and if you query the SimBiology root, there is one model on the root object.

```
rootObj = sbioroot
```

```
SimBiology Root Contains:
```

Models:	1
Builtin Abstract Kinetic Laws:	3
User Abstract Kinetic Laws:	0
Builtin Units:	54
User Units:	0
Builtin Unit Prefixes:	13
User Unit Prefixes:	0

- 2 Use `clear all` to clear the workspace. The `modelObj` still exists on the `rootObj`.

```
clear all
```

```
rootObj
```

```
SimBiology Root Contains:
```

Models:	1
Builtin Abstract Kinetic Laws:	3
User Abstract Kinetic Laws:	0
Builtin Units:	54
User Units:	0
Builtin Unit Prefixes:	13
User Unit Prefixes:	0

- 3 Use `sbioreset` to delete the `modelObj` from the root.

```
sbioreset
```

```
rootObj
```

```
SimBiology Root Contains:
```

Models:	0
Builtin Abstract Kinetic Laws:	3

User Abstract Kinetic Laws:	0
Builtin Units:	54
User Units:	0
Builtin Unit Prefixes:	13
User Unit Prefixes:	0

## See Also

[sbioaddtolibrary](#) | [sbioroot](#) | [sbiounit](#) | [sbiounitprefix](#)

## Topics

[sbioroot](#) on page 1-253

**Introduced before R2006a**

## sbioroot

Return SimBiology root object

### Syntax

```
rootObj = sbioroot
```

### Arguments

<i>rootObj</i>	Return <code>sbioroot</code> to this object.
----------------	--

### Description

`rootObj = sbioroot` returns the SimBiology root object to `root`. The SimBiology root object contains a list of the SimBiology model objects, available units, unit prefixes, and available kinetic laws.

The units define the set of built-in units and user-defined units. See `Unit` object for more information.

The unit prefixes define the set of built-in prefixes and user-defined prefixes. See `Unit Prefix` object for more information.

The kinetic laws define the built-in kinetic laws and user-defined kinetic laws. See `AbstractKineticLaw` object for more information.

To add a unit, prefix or kinetic law to the root (in the user-defined library), use the `sbioaddtolibrary` function. To remove, use `sbioremovefromlibrary`.

The models opened in the SimBiology desktop are stored in the root object.

## Method Summary

copyobj (any object)	Copy SimBiology object and its children
get (any object)	Get object properties
reset (root)	Delete all model objects from root object
set (any object)	Set object properties

## Property Summary

BuiltInLibrary	Library of built-in components
Models	Contain all model objects
Type	Display SimBiology object type
UserDefinedLibrary	Library of user-defined components

## See Also

`Unit object` | `UnitPrefix object` | `addkineticlaw` | `sbiomodel` | `sbioreset`

## Topics

`sbiomodel` on page 1-149  
`addkineticlaw` on page 2-34  
`sbioreset` on page 1-250

**Introduced in R2006a**

# sbiosampleparameters

Generate parameters by sampling covariate model (requires Statistics and Machine Learning Toolbox software)

---

**Note** Support for specifying a numeric vector for `thetas` (fixed effects) or a matrix for `omega` (covariance matrix of random effects) will be removed in a future release. Use a table instead.

---

## Syntax

```
phi = sbiosampleparameters(covexpr,thetas,omega,ds)
phi = sbiosampleparameters(covexpr,thetas,omega,n)
[phi,covmodel] = sbiosampleparameters(_)
```

## Description

`phi = sbiosampleparameters(covexpr,thetas,omega,ds)` generates a matrix `phi` containing sampled parameter values using the covariate model specified by the covariate expression `covexpr`, fixed effects `thetas`, covariance matrix `omega`, and covariate data `ds`.

`phi = sbiosampleparameters(covexpr,thetas,omega,n)` uses a scalar `n` that specifies the number of rows in `phi` when the parameters are not dependent on any covariate.

`[phi,covmodel] = sbiosampleparameters(_)` returns a matrix `phi` and a covariate model object `covmodel` using any of the input arguments from previous syntaxes.

## Examples

### Sample Parameter Values from a Covariate Model

This example uses data collected on 59 preterm infants given phenobarbital during the first 16 days after birth. Each infant received an initial dose followed by one or more

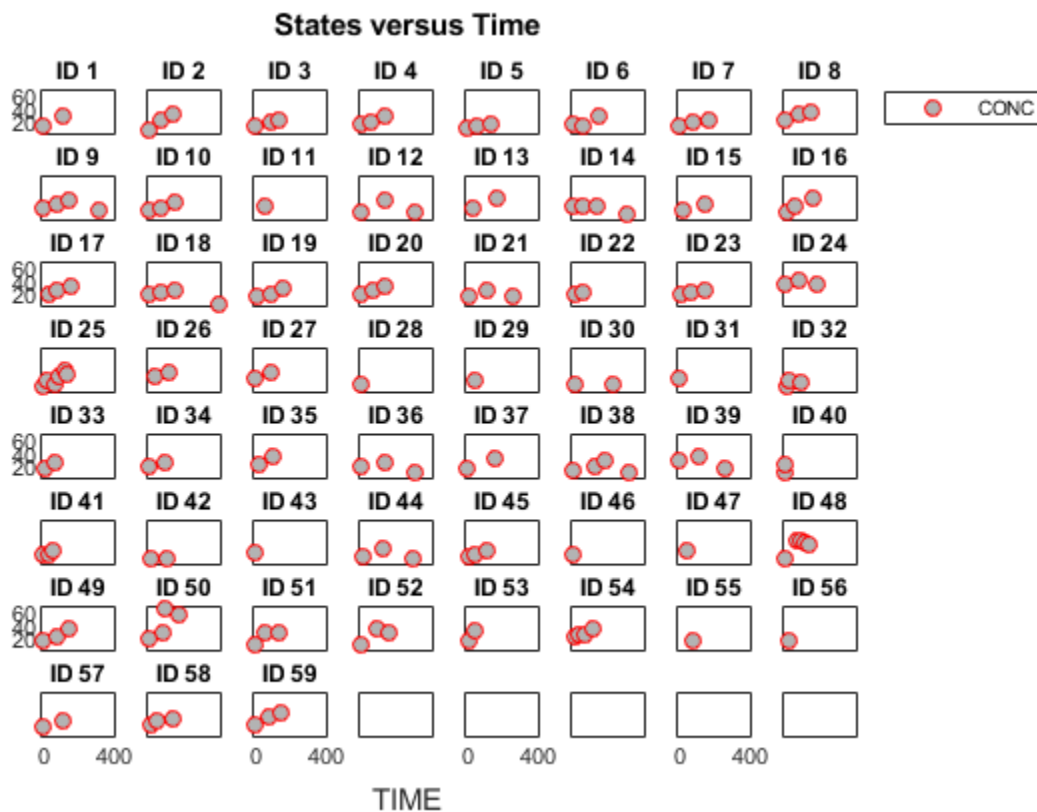
sustaining doses by intravenous bolus administration. A total of between 1 and 6 concentration measurements were obtained from each infant at times other than dose times, for a total of 155 measurements. Infant weights and APGAR scores (a measure of newborn health) were also recorded. Data was described in [1], a study funded by the NIH/NIBIB grant P41-EB01975.

Load the data.

```
load pheno.mat ds
```

Visualize the data.

```
t = sbiotrellis(ds, 'ID', 'TIME', 'CONC', 'marker', 'o', 'markerfacecolor', [.7 .7 .7], 'markerfacealpha', .5, 't.plottitle' = 'States versus Time');
```



Create a one-compartment PK model with bolus dosing and linear clearance to model such data.

```
pkmd = PKModelDesign;
pkmd.addCompartment('Central','DosingType','Bolus','EliminationType','linear-clearance
                    'HasResponseVariable',true,'HasLag',false);
onecomp = pkmd.construct;
```

Suppose there is a correlation between the volume of the central compartment (Central) and the weight of infants. You can define this parameter-covariate relationship using a covariate model that can be described as

$$\log(V_i) = \theta_V + \theta_{V_i} / \text{WEIGHT}_i * \text{WEIGHT}_i + \eta_{V,i}$$

where, for each *i*th infant, *V* is the volume,  $\theta$ s (thetas) are fixed effects,  $\eta$  (eta) represents random effects, and *WEIGHT* is the covariate.

```
covM = CovariateModel;
covM.Expression = {'Central = exp(theta1+theta2*WEIGHT+eta1)'};
```

Define the fixed and random effects. The column names of each table must have the names of fixed effects and random effects, respectively.

```
thetas = table(1.4,0.05,'VariableNames',{'theta1','theta2'});
eta1 = table(0.2,'VariableNames',{'eta1'});
```

Change the group label ID to *GROUP* as required by the *sbiosampleparameters* function.

```
ds.Properties.VariableNames{'ID'} = 'GROUP';
```

Generate parameter values for the volumes of central compartments *Central* based on the covariate model for all infants in the data set.

```
phi = sbiosampleparameters(covM.Expression,thetas,eta1,ds);
```

You can then simulate the model using the sampled parameter values. For convenience, use the function-like interface provided by a *SimFunction* object.

First, construct a *SimFunction* object using the *createSimFunction* method, specifying the volume (*Central*) as the parameter, and the drug concentration in the compartment (*Drug\_Central*) as the output of the *SimFunction* object, and the dosed species.

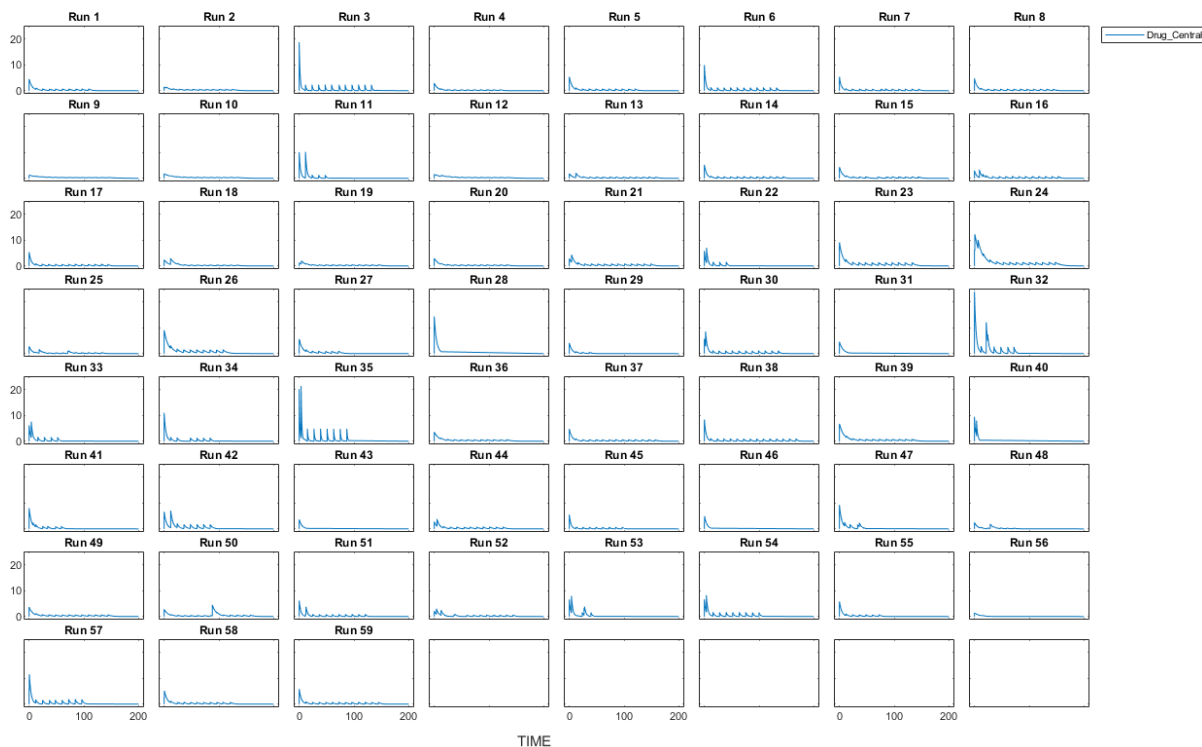
```
f = createSimFunction(onecomp, covM.ParameterNames, 'Drug_Central', 'Drug_Central');
```

The data set `ds` contains dosing information for each infant, and the `groupedData` object provides a convenient way to extract such dosing information. Convert `ds` to a `groupedData` object and extract dosing information.

```
grpData = groupedData(ds);
doses = createDoses(grpData, 'DOSE');
```

Simulate the model using the sampled parameter values from `phi` and the extracted dosing information of each infant, and plot the results. The `ith` run uses the `ith` parameter value in `phi` and dosing information of the `ith` infant.

```
t = sbiotrellis(f(phi,200,doses.getTable), [], 'TIME', 'Drug_Central');
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```





## Input Arguments

### **covexpr** — Covariate expressions

cell array of character vectors | string vector

Covariate expressions, specified as a cell array of character vectors or string vector that defines the parameter-covariate relationships. See `CovariateModel` object to learn more about covariate expressions.

### **thetas** — Fixed effects

table | dataset | numeric vector

Fixed effects, specified as a table, dataset, or numeric vector containing values for fixed effect parameters defined in the covariate expressions `covexpr`. Fixed effect parameter names must start with 'theta'.

- If `thetas` is a table, `thetas.Properties.VariableNames` must match the names of the fixed effects.

For example, suppose that you have three *thetas*:  $\theta_1 = 0.1$ ,  $\theta_2 = 0.2$ , and  $\theta_3 = 0.3$ . You can create the corresponding table.

```
thetas = table(0.1,0.2,0.3);
thetas.Properties.VariableNames = {'thetaOne', 'theta2', 'theta3'}
```

```
thetas =
```

```
1×3 table
```

thetaOne	theta2	theta3
0.1	0.2	0.3

- If `thetas` is a dataset, `thetas.Properties.VarNames` must match the names of the fixed effects.
- If `thetas` is a numeric vector, the order of the values in the vector must be the same ascending ASCII dictionary order as the fixed effect names.

Use the `sort` function to sort a cell array of character vectors to see the order.

```
sort({'thetaOne', 'theta2', 'theta3'})
```

```
ans =
    1×3 cell array
    {'theta2'}    {'theta3'}    {'thetaOne'}
```

Then specify the value of each *theta* in the same order.

```
thetas = [0.2 0.3 0.1];
```

### omega — Covariance matrix of random effects

table | dataset | matrix

Covariance matrix of random effects, specified as a table, dataset, or matrix. Random effect parameter names must start with 'eta'.

- If *omega* is a table, *omega.Properties.VariableNames* must match the names of the random effects. Specifying the row names (*RowNames*) is optional, but if you do, they must also match the names of random effects.

Suppose that you want to define a diagonal covariance matrix with three random effect parameters *eta1*, *eta2*, and *eta3* with the values 0.1, 0.2, and 0.3, respectively.

$$\begin{bmatrix} Cov(\eta_1, \eta_1) & Cov(\eta_1, \eta_2) & Cov(\eta_1, \eta_3) \\ Cov(\eta_2, \eta_1) & Cov(\eta_2, \eta_2) & Cov(\eta_2, \eta_3) \\ Cov(\eta_3, \eta_1) & Cov(\eta_3, \eta_2) & Cov(\eta_3, \eta_3) \end{bmatrix} = \begin{bmatrix} eta1 & 0 & 0 \\ 0 & eta2 & 0 \\ 0 & 0 & eta3 \end{bmatrix}$$

You can construct the corresponding table.

```
eta1 = [0.1;0;0];
eta2 = [0;0.2;0];
eta3 = [0;0;0.3];
omega = table(eta1,eta2,eta3,'VariableNames',{'eta1','eta2','eta3'})
```

```
omega =
    3×3 table
    eta1    eta2    eta3
    ---    ---    ---
    0.1      0      0
```

```

      0      0.2      0
      0      0      0.3

```

- If `omega` is a dataset, `omega.Properties.VarNames` must match the names of the random effects. Specifying the row names (`ObsNames`) is optional, but if you do, they must also match the names of random effects.
- If `omega` is a matrix, the rows and columns must have the same ascending ASCII dictionary order as the random effect names.

Use the `sort` function to sort a cell array of character vectors to see the order.

```

sort({'eta1', 'eta2', 'eta3'})

ans =

1x3 cell array

    {'eta1'}    {'eta2'}    {'eta3'}

```

### **ds — Covariate data**

dataset | table

Covariate data, specified as a dataset or table containing the covariate data for all groups.

`ds` must have a column named 'Group' or 'GROUP' specifying the group labels as well as a column each for all covariates used in the covariate model. The column names must match the names of the corresponding covariates used in the covariate expressions.

### **n — Number of rows in phi**

scalar

Number of rows in `phi`, specified as a scalar.

## **Output Arguments**

### **phi — Sampled parameter values**

matrix

Sampled parameter values, returned as a matrix of size S-by-P, where S is the number of groups specified in `ds` or specified by `n` and P is the number of parameters which is equal to the number of elements in `covexpr`.

## **covmodel — Covariate model**

CovariateModel object

Covariate model, returned as a `CovariateModel` object which represents the model defined by `covexpr`.

## **References**

- [1] Grasela Jr, T.H., Donn, S.M. (1985) Neonatal population pharmacokinetics of phenobarbital derived from routine clinical data. *Dev Pharmacol Ther.* 8(6), 374-83.

## **See Also**

`CovariateModel` object | `SimFunction` object | `createSimFunction` | `sbiosampleerror`

**Introduced in R2014a**

# sbiosampleerror

Sample error based on error model and add noise to simulation data

## Syntax

```
sdN = sbiosampleerror(sd,errormodel,errorparam)
```

## Description

`sdN = sbiosampleerror(sd,errormodel,errorparam)` adds noise to the simulation data `sd` using one or more error models `errormodel` and error parameters `errorparam`.

## Examples

### Add Noise to Simulation Data

This example adds noise (or error) to the simulation data from a radioactive decay model

with the first-order reaction:  $\frac{dz}{dt} = c \cdot x$ , where `x` and `z` are species and `c` is the forward rate constant.

Load the sample project containing the radiodecay model `m1`.

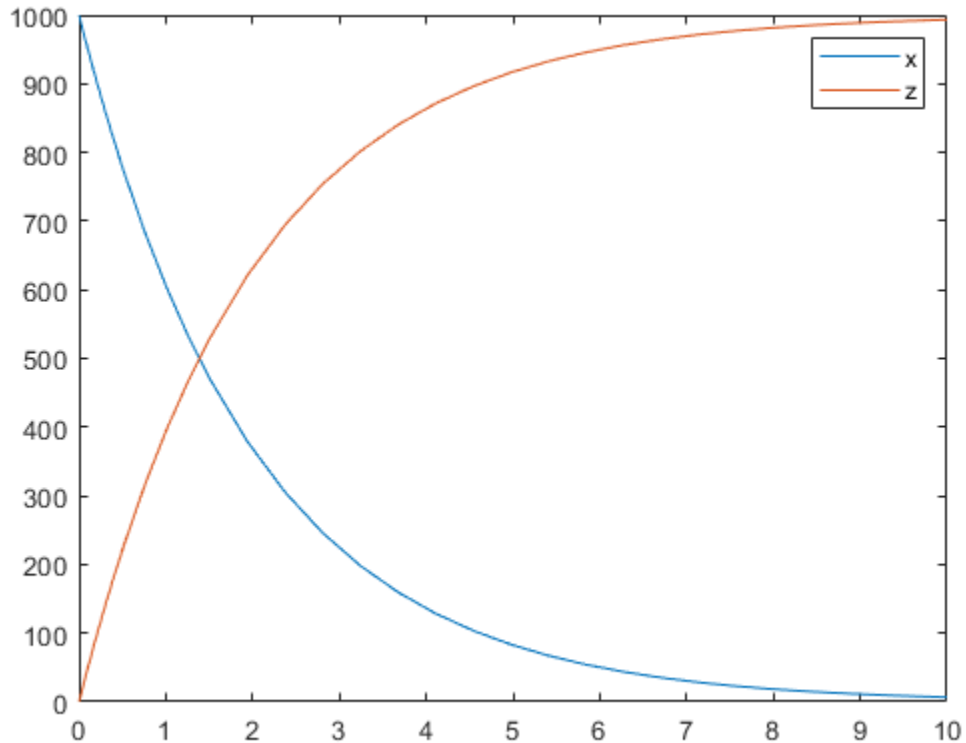
```
sbioloadproject radiodecay;
```

Simulate the model.

```
[t,sd,names] = sbiosimulate(m1);
```

Plot the simulation results.

```
plot(t,sd);  
legend(names,'AutoUpdate','off');  
hold on
```

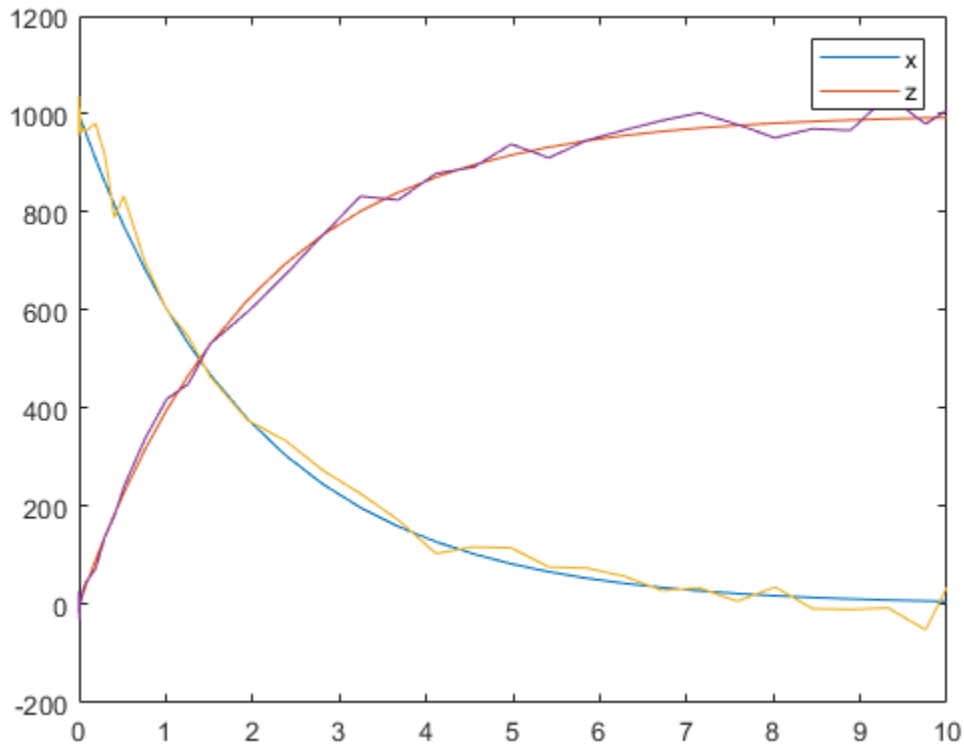


Add noise to the simulation results using the constant error model with the error parameter set to 20.

```
sdNoisy = sbiosampleerror(sd,'constant',20);
```

Plot the noisy simulation data.

```
plot(t,sdNoisy);
```



### Define a Custom Error Model Using a Function Handle

This example defines a custom error model using a function handle and adds noise to simulation data of a radioactive decay model with the first-order reaction  $\frac{dz}{dt} = c \cdot x$ , where  $x$  and  $z$  are species, and  $c$  is the forward rate constant.

Load the sample project containing the radiodecay model `m1`.

```
sbioloadproject radiodecay;
```

Suppose you have a simple custom error model with a standard mean-zero and unit-variance (Gaussian) normal variable  $e$ , simulation results  $f$ , and two parameters  $p1$  and  $p2$ :  $y = f + p1 + p2 * e$

Define a function handle that represents the error model.

```
em = @(y,p1,p2) y+p1+p2*randn(size(y));
```

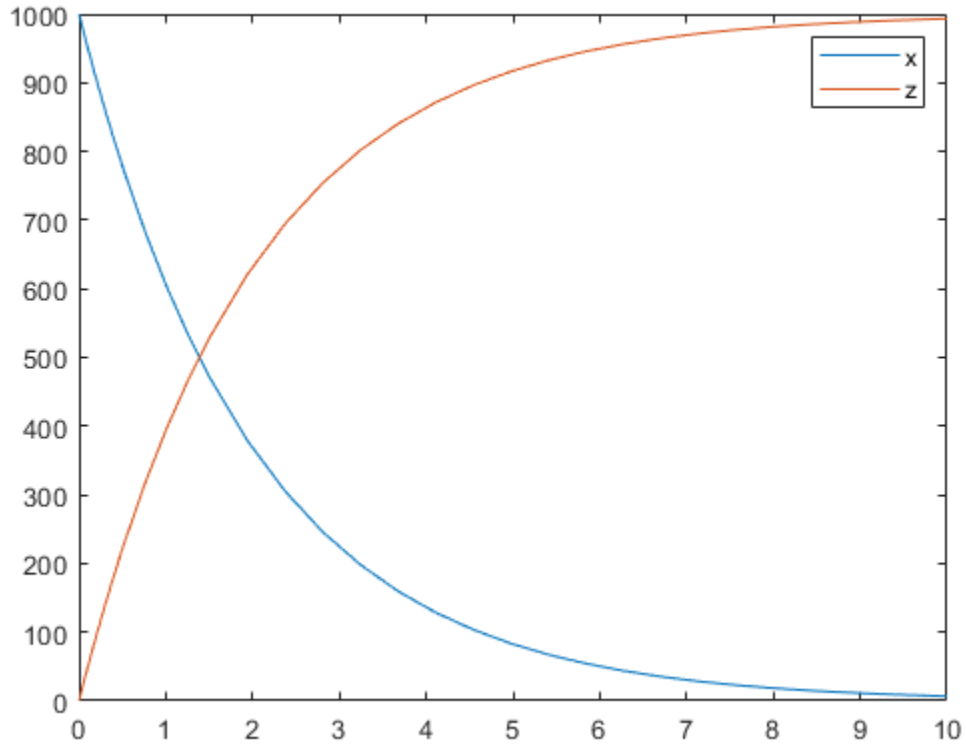
Simulate the model.

```
[t,sd,names] = sbiosimulate(m1);
```

Plot the simulation results and hold the plot.

```
plot(t,sd);  
legend(names, 'AutoUpdate', 'off');  
hold on
```



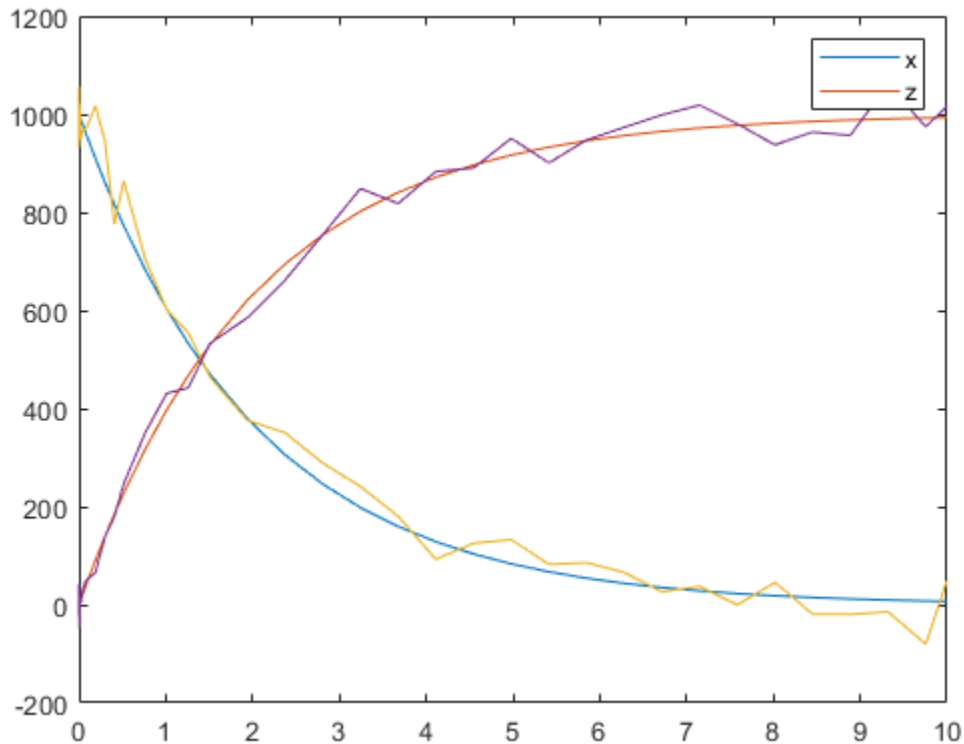


Sample the error using the previously defined custom function with two parameters set to 0.5 and 30, respectively.

```
sdNoisy = sbiosampleerror(sd,em,{0.5,30});
```

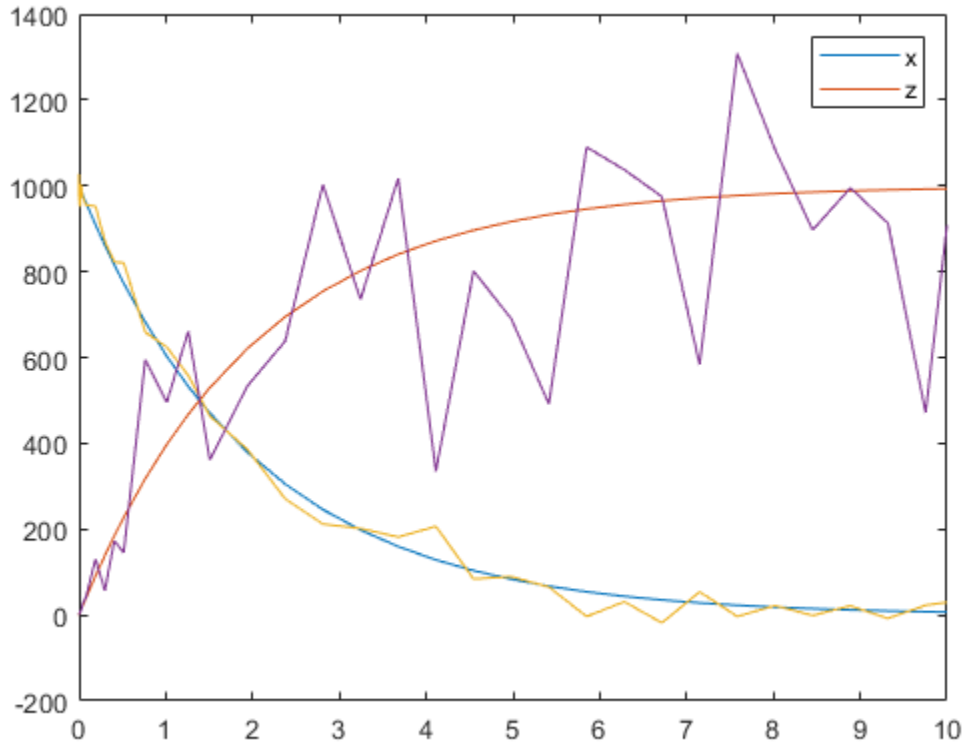
Plot the noisy simulation data.

```
plot(t,sdNoisy);
```



You can also apply a different error model to each state, which is a column in `sd`. Suppose you want to apply the custom error model (`em`) to the first column (species `x` data) and the proportional error model to the second column (species `z` data).

```
hold off
sdNoisy = sbiosampleerror(sd,{em,'proportional'},{{0.5,30},0.3});
plot(t,sd);
legend(names,'AutoUpdate','off');
hold on
plot(t,sdNoisy);
```



## Input Arguments

### **sd** — Simulation results

`SimData` object | matrix

Simulation results, specified as a `SimData` object or matrix.

### **errormodel** — Error model

character vector | string | function handle | string vector | cell array of character vectors

Error model(s), specified as a character vector, string, function handle, string vector, cell array of character vectors, or cell array containing a mixture of character vectors and function handles.

If it is a string vector or cell array, its length must match the number of columns (responses) in `sd`, and each error model is applied to the corresponding column in `sd`. If it is a single character vector, string, or function handle, the same error model is applied to all columns in `sd`.

The first argument of a function handle must be a matrix of simulation results. The subsequent arguments are the parameters of the error model supplied in the `errorparam` input argument. The output of the function handle must be a matrix of the same size as the first input argument (simulation results).

For example, suppose you have a custom error model with a standard mean-zero and unit-variance (Gaussian) normal variable  $e$ , simulation results  $f$ , and two parameters  $p1$  and  $p2$ :  $y = f + p1 + p2 * e$ . You can define the corresponding function handle as follows.

```
em = @(y,p1,p2) y+p1+p2*randn(size(y));
```

where  $y$  is the matrix of simulation results and  $p1$  and  $p2$  are the error parameters. The output of the function handle must be the same size as  $y$ , which is the same as the simulation results specified in the `sd` input argument. The parameters  $p1$  and  $p2$  are specified in the `errorparam` argument.

There are four built-in error models. Each model defines the error using a standard mean-zero and unit-variance (Gaussian) variable  $e$ , simulation results  $f$ , and one or two parameters  $a$  and  $b$ . The models are:

- 'constant':  $y = f + ae$
- 'proportional':  $y = f + b|f|e$
- 'combined':  $y = f + (a + b|f|)e$
- 'exponential':  $y = f * \exp(ae)$

**errorparam — Error model parameter**

scalar | vector | cell array

Error model parameter(s), specified as a scalar, vector, or cell array. If `errormodel` is 'constant', 'proportional', or 'exponential', then `errorparam` is specified as a

numeric scalar. If it is 'combined', then `errorparam` is specified as a row vector with two elements [a b].

If `errormodel` is a cell array, then `errorparam` must be a cell array of the same length. In other words, `errorparam` must contain N elements, where N is the number of error models in `errormodel`. Each element must have the correct number of parameters for the corresponding error model.

For example, suppose you have three columns in `sd`, and you are applying a different error model (constant, proportional, and exponential error models with error parameters 0.1, 2, and 0.5, respectively) to each column, then `errormodel` and `errorparam` must be cell arrays with three elements as follows.

```
errormodel = {'constant', 'proportional', 'exponential'};  
errorparam = {0.1, 2, 0.5};
```

## Output Arguments

### **sdN — Simulation results with added noise**

vector of `SimData` objects | matrix

Simulation results with added noise, returned as a vector of `SimData` objects or numeric matrix. If `sd` is a vector of `SimData` objects, `sdN` is also a vector of `SimData` objects, and the error is added to each column in the `sd.Data` property. If `sd` is specified as a matrix, `sdN` is a matrix, and the error is added to each column in the matrix.

## See Also

`SimFunction` object | `createSimFunction` | `sbiosampleparameters`

**Introduced in R2014a**

## sbiosaveproject

Save all models in root object

### Syntax

```
sbiosaveproject projFilename  
sbiosaveproject projFilename variableName  
sbiosaveproject projFilename variableName1 variableName2 ...
```

### Description

`sbiosaveproject projFilename` saves all models in the SimBiology root object to the binary SimBiology project file named `projFilename.sbproj`. The project can be loaded with `sbioloadproject`. `sbiosaveproject` returns an error if `projFilename.sbproj` is not writable.

`sbiosaveproject` creates the binary SimBiology project file named `simbiology.sbproj`. `sbiosaveproject` returns an error if this is not writable.

`sbiosaveproject projFilename variableName` saves only `variableName`. `variableName` can be a SimBiology model or any MATLAB variable.

`sbiosaveproject projFilename variableName1 variableName2 ...` saves the specified variables in the project.

Use the functional form of `sbiosaveproject` when the file name or variable names are stored in a character vector. For example, if the file name is stored in the variable `fileName` and you want to store MATLAB variables `variableName1` and `variableName2`, type `sbiosaveproject (projFileName, 'variableName1', 'variableName2')` at the command line.

### Examples

- 1 Import an SBML file and simulate (default configset object is used).

```
modelObj = sbmlimport ('oscillator.xml');  
timeseriesObj = sbiosimulate(modelObj);
```

- 2 Save the model and the simulation results to a project.

```
sbiosaveproject myprojectfile modelObj timeseriesObj
```

## See Also

[sbioaddtolibrary](#) | [sbioloadproject](#) | [sbioremovefromlibrary](#) | [sbiowhos](#)

## Topics

[sbioloadproject](#) on page 1-147

[sbiowhos](#) on page 1-340

[sbioaddtolibrary](#) on page 1-22

[sbioremovefromlibrary](#) on page 1-248

## Introduced in R2006a

## sbioselect

Search for objects with specified constraints

### Syntax

```

Out = sbioselect('PropertyName', PropertyValue)
Out = sbioselect('Where', 'PropertyName', 'Condition',
    PropertyValue)
Out = sbioselect(Obj, 'PropertyName', PropertyValue)
Out = sbioselect(Obj, 'Type', 'TypeValue', 'PropertyName',
    PropertyValue)
Out = sbioselect(Obj, 'Where', 'PropertyName', 'Condition',
    PropertyValue)
Out = sbioselect(Obj, 'Where', 'PropertyNameCondition',
    'PropertyNamePattern', 'Condition', PropertyValue)
Out = sbioselect(Obj, 'Where', 'PropertyName1', 'Condition1',
    PropertyValue1, 'Where', 'PropertyName2', 'Condition2',
    PropertyValue2,...)
Out = sbioselect(Obj, 'Where', 'PropertyName1', 'Condition1',
    PropertyValue1, Bool_Operator, 'Where', 'PropertyName2',
    'Condition2', PropertyValue2,...)
Out = sbioselect(Obj, 'Depth', DepthValue,...)
    
```

### Arguments

<i>Out</i>	Object or array of objects returned by the <code>sbioselect</code> function. <i>Out</i> might contain a mixture of object types (for example, species and parameters), depending on the selection you specify.  If <i>PropertyValue</i> is a cell array, then the function returns all objects with the property ' <i>PropertyName</i> ' that matches any element of <i>PropertyValue</i> .
<i>Obj</i>	SimBiology object or array of objects to search. If an object is not specified, <code>sbioselect</code> searches the root.



<i>PropertyName</i>	Any property of the object being searched.
<i>PropertyValue</i>	Specify <i>PropertyValue</i> to include in the selection criteria.
<i>TypeValue</i>	Type of object to include in the selection, for example, <code>sbiomodel</code> , <code>species</code> , <code>reaction</code> , or <code>kineticlaw</code> .
<i>Condition</i>	The search condition. See the table under “Description” on page 1-275 for a list of conditions.
<i>PropertyNameCondition</i>	Search condition that applies only to the name property. See the table listing “Conditions for Names” below.
<i>PropertyNamePattern</i>	Character vector or string used to select the property name according to the condition imposed by <i>PropertyNameCondition</i> .
<i>DepthValue</i>	Specify the depth number to search. Valid numbers are positive integer values and <code>inf</code> . If <i>DepthValue</i> is <code>inf</code> , <code>sbioselect</code> searches <i>Obj</i> and all of its children. If <i>DepthValue</i> is <code>1</code> , <code>sbioselect</code> only searches <i>Obj</i> and not its children. By default, <i>DepthValue</i> is <code>inf</code> .

## Description

`sbioselect` searches for objects with specified constraints.

`Out = sbioselect('PropertyName', PropertyValue)` searches the root object (including all model objects contained by the root object) and returns the objects with the property name (*PropertyName*) and property value (*PropertyValue*) contained by the root object.

`Out = sbioselect('Where', 'PropertyName', 'Condition', PropertyValue)` searches the root object and finds objects that have a property name (*PropertyName*) and value (*PropertyValue*) that matches the condition (*Condition*).

`Out = sbioselect(Obj, 'PropertyName', PropertyValue)` returns the objects with the property name (*PropertyName*) and property value (*PropertyValue*) found in any object (*Obj*). If the property name in a property-value pair contains either a '?' or '\*', then the name is automatically interpreted as a wildcard expression, equivalent to the where clause (`'Where', 'wildcard', 'PropertyName', '==', PropertyValue`).

`Out = sbioselect(Obj, 'Type', 'TypeValue', 'PropertyName', PropertyValue)` finds the objects of type (*TypeValue*), with the property name

(*PropertyName*) and property value (*PropertyValue*) found in any object (*Obj*). *TypeValue* is the type of SimBiology object to be included in the selection, for example, species, reaction, or kineticlaw.

*Out* = `sbioreselect(Obj, 'Where', 'PropertyName', 'Condition', PropertyValue)` finds objects that have a property name (*PropertyName*) and value (*PropertyValue*) that match the condition (*Condition*).

If you search for a character vector property value without specifying a condition, you must use the same format as `get` returns. For example, if `get` returns the Name as 'MyObject', `sbioreselect` will not find an object with a Name property value of 'myobject'. Therefore, for this example, you must specify:

```
modelObj = sbioreselect ('Name', 'MyObject')
```

Instead, if you use a condition, you can specify:

```
modelObj = sbioreselect ('Where', 'Name', '==i', 'myobject')
```

Thus, conditions let you control the specificity of your selection.

`sbioreselect` searches for model objects on the root in both cases.

*Out* = `sbioreselect(Obj, 'Where', 'PropertyNameCondition', 'PropertyNamePattern', 'Condition', PropertyValue)` finds objects with a property name that matches the pattern in (*PropertyNamePattern*) with the condition (*PropertyNameCondition*) and matches the value (*PropertyValue*) with the condition (*Condition*). Use this syntax when you want search conditions on both property names and property values.

The conditions, with examples of property names and corresponding examples of property values that you can use, are listed in the following tables. This table shows you conditions for numeric properties.

Conditions for Numeric Properties	Example Syntax
==	<p>Search in the model object (<code>modelObj</code>), and return parameter objects that have <code>Value</code> equal to 0.5. <code>sbioselect</code> returns parameter objects because only parameter objects have a property called <code>Value</code>.</p> <pre>parameterObj = sbioselect (modelObj,...   'Where', 'Value', '==', 0.5)</pre> <p>In the case of <code>==</code>, this is equivalent to omitting the condition as shown:</p> <pre>parameterObj = sbioselect (modelObj,...   'Value', 0.5)</pre> <p>Search in the model object (<code>modelObj</code>), and return parameter objects that have <code>ConstantValue</code> false (nonconstant parameters).</p> <pre>parameterObj = sbioselect (modelObj,...   'Where', 'ConstantValue', '==', false)</pre>
~=	<p>Search in the model object (<code>modelObj</code>), and return parameter objects that do not have <code>Value</code> equal to 0.5.</p> <pre>parameterObj = sbioselect (modelObj,...   'Where', 'Value', '~=', 0.5)</pre>
>,<,>=,<=	<p>Search in the model object (<code>modelObj</code>), and return species objects that have an initial amount (<code>InitialAmount</code>) greater than 50.</p> <pre>speciesObj = sbioselect (modelObj, ...   'Where', 'InitialAmount', '&gt;', 50)</pre> <p>Search in the model object (<code>modelObj</code>), and return species objects that have an initial amount (<code>InitialAmount</code>) less than or equal to 50.</p> <pre>speciesObj = sbioselect (modelObj,...   'Where', 'InitialAmount', '&lt;=', 50)</pre>

Conditions for Numeric Properties	Example Syntax
between	<p>Search in the model object (<code>modelObj</code>), and return species objects that have an initial amount (<code>InitialAmount</code>) between 200 and 300.</p> <pre>speciesObj = sbioselect (modelObj,...     'Where', 'InitialAmount',...     'between', [200 300])</pre>
~between	<p>Search in the model object (<code>modelObj</code>), and return species objects that have an initial amount (<code>InitialAmount</code>) that is not between 200 and 300.</p> <pre>speciesObj = sbioselect (modelObj,...     'Where', 'InitialAmount',...     '~between', [200 300])</pre>
equal_and_same_type	<p>Similar to <code>==</code>, but in addition requires the property value to be of the same type.</p> <p>Search in the model object (<code>modelObj</code>), and return all objects containing a property of type <code>double</code> and a value equal to 0. (Using <code>'=='</code> would also select objects containing a property with a value of <code>false</code>.)</p> <pre>zeroObj = sbioselect(modelObj, ...     'Where', '*', 'equal_and_same_type', 0);</pre>
unequal_and_same_type	<p>Similar to <code>~=</code>, but in addition requires the property value to be of the same type.</p> <p>Select all objects containing a property of type <code>double</code> and value not equal to 0. (Using <code>'~='</code> would also select objects containing a property with a value of <code>true</code>.)</p> <pre>nonzeroObj = sbioselect(modelObj, ...     'Where', '*', 'unequal_and_same_type', 0);</pre>

The following table shows you conditions for the name property or for properties whose values are character vectors.

Conditions for Names	Example Syntax
==	<p>Search in the model object (<code>modelObj</code>), and return species objects named 'Glucose'.</p> <pre>speciesObj = sbioselect (modelObj,...   'Type', 'species', 'Where',...   'Name', '==', 'Glucose')</pre>
~=	<p>Search in the model object (<code>modelObj</code>), and return species objects that are not named 'Glucose'.</p> <pre>speciesObj = sbioselect (modelObj,...   'Type', 'species', 'Where',...   'Name', '~=', 'Glucose')</pre>
==i	<p>Same as ==; in addition, this is case insensitive.</p>
~=i	<p>Search in the model object (<code>modelObj</code>), and return species objects that are not named 'Glucose', ignoring case.</p> <pre>speciesObj = sbioselect (modelObj,...   'Type', 'species', 'Where',...   'Name', '~=i', 'glucose')</pre>
<p><code>regexp</code>. Supports expressions supported by the functions <code>regexp</code> and <code>regexpi</code>.</p>	<p>Search in the model object (<code>modelObj</code>), and return objects that have 'ese' or 'ase' anywhere within the name.</p> <pre>Obj = sbioselect (modelObj, 'Where',...   'Name', 'regexp', '[ea]se')</pre> <p>Search in the root, and return objects that have kinase anywhere within the name.</p> <pre>Obj = sbioselect ('Where',...   'Name', 'regexp', 'kinase')</pre> <p>Note that this query could result in a mixture of object types (for example, species and parameters).</p>
regexpi	<p>Same as <code>regexp</code>; in addition, this is case insensitive.</p>
~regexp	<p>Search in the model object (<code>modelObj</code>), and return objects that do not have kinase anywhere within the name.</p> <pre>Obj = sbioselect (modelObj, 'Where',...   'Name', '~regexp', 'kinase')</pre>

Conditions for Names	Example Syntax
~regexp <i>i</i>	Same as ~ <i>regexp</i> ; in addition, this is case insensitive.
wildcard	Supports DOS-style wildcards ('?' matches any single character, '*' matches any number of characters, and the pattern must match the entire character vector). See <i>regexprtranslate</i> for more information.
wildcard <i>i</i>	Same as <i>wildcard</i> ; in addition, this is case insensitive.
~wildcard	Search in the model object ( <i>modelObj</i> ), and return objects that have names that do not begin with <i>kin*</i> .  Obj = <i>sbiomselect</i> ( <i>modelObj</i> , 'Where', ... 'Name', '~wildcard', 'kin*')
~wildcard <i>i</i>	Same as ~ <i>wildcard</i> ; in addition, this is case insensitive.

Use the condition type function for any property. The specified value should be a function handle that, when applied to a property value, returns a boolean indicating whether there is a match. The following table shows an example of using function.

Condition	Example Syntax
'function'	Search in the model object and return reaction objects whose <i>Stoichiometry</i> property contains the specified stoichiometry.  Out = <i>sbiomselect</i> ( <i>modelObj</i> , 'Where', ... 'Stoichiometry', 'function', ... @(x)any(x>2))  Select all objects with a numeric value that is even.  iseven = @(x) isnumeric(x)... && isvector(x) && mod(x, 2) == 0; evenValuedObj = <i>sbiomselect</i> ( <i>modelObj</i> , ... 'where', 'Value', 'function', iseven);

The condition 'contains' can be used only for those properties whose values are an array of SimBiology objects. The following table shows an example of using contains.

Condition	Example Syntax
'contains'	<p>Search in the model object and return reaction objects whose Reactant property contains the specified species.</p> <pre>Out = sbioselect(modelObj, 'Where', ... 'Reactants', 'contains', ... modelObj.Species(1))</pre>

*Out = sbioselect(Obj, 'Where', 'PropertyName1', 'Condition1', PropertyValue1, 'Where', 'PropertyName2', 'Condition2', PropertyValue2, ...)* finds objects contained by *Obj* that matches all the conditions specified.

You can combine any number of property name/property value pairs and conditions in the *sbioselect* command.

*Out = sbioselect(Obj, 'Where', 'PropertyName1', 'Condition1', PropertyValue1, Bool\_Operator, 'Where', 'PropertyName2', 'Condition2', PropertyValue2, ...)* finds objects contained by *Obj* that matches all the conditions specified. Supported character vectors for *Bool\_Operator* are as follows.

- 'and' True if ( 'Where', 'PropertyName1', 'Condition1', PropertyValue1) and ( 'Where', 'PropertyName2', 'Condition2', PropertyValue2) are both true.
- 'or' True if either ( 'Where', 'PropertyName1', 'Condition1', PropertyValue1) or ( 'Where', 'PropertyName2', 'Condition2', PropertyValue2) is true.
- 'xor' True if exactly one of ( 'Where', 'PropertyName1', 'Condition1', PropertyValue1) or ( 'Where', 'PropertyName2', 'Condition2', PropertyValue2) is true.
- 'not' True if ( 'Where', 'PropertyName1', 'Condition1', PropertyValue1) is true and ( 'Where', 'PropertyName2', 'Condition2', PropertyValue2) is not true.

Compound expressions with multiple boolean operators are supported. Precedence of the operators follows the order of operations for boolean algebra not -> and -> xor -> or.

*Out = sbioselect(Obj, 'Depth', DepthValue, ...)* finds objects using a model search depth of *DepthValue*.

**Note** The order of results from `sbioselect` is not guaranteed when returning results of multiple types. Hence it is not recommended to depend on the order of results.

---

## Examples

### Find Species from a SimBiology Model

Import a model.

```
modelObj = sbmlimport('oscillator');
```

Find and return an object named `pA`.

```
pA = sbioselect(modelObj, 'Name', 'pA')
```

SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	pA	100	

Find and return species objects whose `Name` starts with `p` and have `A` or `B` as the next letter in the name.

```
speciesObjs = sbioselect(modelObj, 'Type', 'species', 'Where', ...  
                        'Name', 'regexp', '^p[AB]')
```

SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	pA	100	
2	unnamed	pB	0	
3	unnamed	pA_0pB1	0	
4	unnamed	pB_0pC1	0	
5	unnamed	pA_0pB_pA	20	
6	unnamed	pA_0pB2	0	
7	unnamed	pB_0pC2	0	
8	unnamed	pB_0pC_pB	0	

Find a cell array. Note how cell array values must be specified inside another cell array.

```
modelObj.Species(2).UserData = {'a' 'b'};  
Obj = sbioselect(modelObj, 'UserData', {'a' 'b'})
```



SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	pB	0	

Find and return objects that do not have their units set.

```
unitlessObj = sbioselect(modelObj, 'Where', 'wildcard', '*Units', '==', '');
```

Alternatively, you can do the following.

```
unitlessObj = sbioselect(modelObj, '*Units', '');
```

## See Also

regexp

**Introduced before R2006a**

## sbioshowunitprefixes

Show unit prefixes in library

### Syntax

```
UnitPrefixObjs = sbioshowunitprefixes  
[Name, Multiplier] = sbioshowunitprefixes  
[Name, Multiplier, Builtin] = sbioshowunitprefixes  
[Name, Multiplier, Builtin] = sbioshowunitprefixes('Name')
```

### Arguments

<i>unitPrefixObjs</i>	Vector of unit prefix objects from the BuiltInLibrary and UserDefinedLibrary properties of the Root.
<i>Name</i>	Name of the built-in or user-defined unit prefix. Built-in prefixes are defined based on the International System of Units (SI).
<i>Multiplier</i>	Shows the value of $10^{\text{Exponent}}$ that defines the relationship of the unit prefix <i>Name</i> to the base unit. For example, the multiplier in picomole is $10e-12$ .
<i>Builtin</i>	An array of logical values. If <i>Builtin</i> is true for a unit prefix, the unit prefix is built in. If <i>Builtin</i> is false for a unit prefix, the unit prefix is user-defined.

### Description

sbioshowunitprefixes returns information about unit prefixes in the SimBiology library.

*UnitPrefixObjs* = sbioshowunitprefixes returns the unit prefixes in the library as a vector of unit prefix objects in *UnitPrefixObjs*.

`[Name, Multiplier] = sbioshowunitprefixes` returns the multiplier for each prefix in *Name* to *Multiplier* as a cell array of character vectors.

`[Name, Multiplier, Builtin] = sbioshowunitprefixes` returns whether the unit prefix is built in or user defined for each unit prefix in *Name* to *Builtin*.

`[Name, Multiplier, Builtin] = sbioshowunitprefixes('Name')` returns the name, multiplier, and built-in status for the unit prefix with name *Name*. *Name* can be a string vector or cell array of character vectors.

## Examples

```
[name, multiplier] = sbioshowunitprefixes;  
[name, multiplier] = sbioshowunitprefixes('nano');
```

## See Also

[sbioconvertunits](#) | [sbioshowunits](#) | [sbiounitprefix](#)

**Introduced in R2006a**

## sbioshowunits

Show units in library

### Syntax

```
unitObjs = sbioshowunits  
[Name, Composition] = sbioshowunits  
[Name, Composition, Multiplier] = sbioshowunits  
[Name, Composition, Multiplier, Offset] = sbioshowunits  
[Name, Composition, Multiplier, Offset, Builtin] = sbioshowunits  
[Name, Composition, Multiplier, Offset, Builtin] =  
sbioshowunits('Name')
```

### Arguments

<i>unitObjs</i>	Vector of unit objects from the BuiltInLibrary and UserDefinedLibrary properties of the Root.
<i>Name</i>	Name of the built-in or user-defined unit.
<i>Composition</i>	Shows the combination of base and derived units that defines the unit <i>Name</i> . For example, molarity is mole/liter.
<i>Multiplier</i>	The numerical value that defines the relationship between the unit <i>Name</i> and the base or derived unit as a product of the <i>Multiplier</i> and the base unit or derived unit. For example, 1 mole is $6.0221e23$ *molecule. The <i>Multiplier</i> is $6.0221e23$ .
<i>Offset</i>	Numerical value by which the unit composition is modified from the base unit. For example, $^{\circ}\text{Celsius} = (5/9)*(^{\circ}\text{Fahrenheit}-32)$ ; <i>Multiplier</i> is 5/9 and <i>Offset</i> is $-32$ .

<i>Builtin</i>	An array of logical values. If <i>Builtin</i> is true for a unit, the unit is built in. If <i>Builtin</i> is false for a unit, the unit is user-defined.
----------------	--

## Description

*unitObjs* = sbioshowunits returns the units in the library to *unitObjs* as a vector of unit objects.

[*Name*, *Composition*] = sbioshowunits returns the composition for each unit in *Name* to *Composition* as a cell array of character vectors.

[*Name*, *Composition*, *Multiplier*] = sbioshowunits returns the multiplier for the unit with name *Name* to *Multiplier*.

[*Name*, *Composition*, *Multiplier*, *Offset*] = sbioshowunits returns the offset for the unit with name *Name* to *Offset*. The unit is defined as  $Multiplier * Composition + Offset$ .

[*Name*, *Composition*, *Multiplier*, *Offset*, *Builtin*] = sbioshowunits returns whether the unit is built in or user defined for each unit in *Name* to *Builtin*.

[*Name*, *Composition*, *Multiplier*, *Offset*, *Builtin*] = sbioshowunits('Name') returns the name, composition, multiplier, offset and built-in status for the unit with name *Name*. *Name* can be a string vector or cell array of character vectors.

## Examples

```
[name, composition] = sbioshowunits;
[name, composition] = sbioshowunits('molecule');
```

## See Also

sbioconvertunits | sbioshowunitprefixes | sbiounit

Introduced in R2006a

## sbiosimulate

Simulate SimBiology model

### Syntax

```
[time,x,names] = sbiosimulate(modelObj)
[time,x,names] = sbiosimulate(modelObj,csObj)
[time,x,names] = sbiosimulate(modelObj,dvObj)
[time,x,names] = sbiosimulate(modelObj,csObj,dvObj)
[time,x,names] = sbiosimulate(modelObj,csObj,variantObj,doseObj)

simDataObj = sbiosimulate( ___ )
```

### Description

`[time,x,names] = sbiosimulate(modelObj)` returns simulation results in three outputs, `time`, vector of time samples, `x`, simulation data, and `names`, column labels of simulation data `x`. This function simulates the SimBiology model `modelObj` while using the active configuration set along with its active doses and active variants if any.

`[time,x,names] = sbiosimulate(modelObj,csObj)` returns simulation results using the specified configset object `csObj`, any active variants, and any active doses. Any other configsets are ignored. If you set `csObj` to empty `[]`, the function uses the active configset.

`[time,x,names] = sbiosimulate(modelObj,dvObj)` returns simulation results using doses or variants specified by `dvObj` and the active configset. `dvObj` can be one of the following:

- Variant object
- ScheduleDose object
- RepeatDose object
- Array of doses or variants

If you set `dvObj` to empty `[]`, the function uses the active configset, active variants, and active doses.

If you specify `dvObj` as variants, the function uses the specified variants and active doses. Any other variants are ignored.

If you specify `dvObj` as doses, the function uses the specified doses and active variants. Any other doses are ignored.

`[time,x,names] = sbiosimulate(modelObj,csObj,dvObj)` returns simulation results using a configset object `csObj` and dose, variant, or an array of doses or variants specified by `dvObj`.

If you set `csObj` to `[]`, then the function uses the active configset object.

If you set `dvObj` to `[]`, then the function uses no variants, but uses active doses.

If you specify `dvObj` as variants, the function uses the specified variants and active doses. Any other variants are ignored.

If you specify `dvObj` as doses, the function uses the specified doses and active variants. Any other doses are ignored.

`[time,x,names] = sbiosimulate(modelObj,csObj,variantObj,doseObj)` returns simulation results using a configset object `csObj`, variant object or variant array specified by `variantObj`, and dose object or dose array specified by `doseObj`.

If you set `csObj` to `[]`, then the function uses the active configset object.

If you set `variantObj` to `[]`, then the function uses no variants.

If you set `doseObj` to `[]`, then the function uses no doses.

`simDataObj = sbiosimulate( __ )` returns simulation results in a `SimData` object `simDataObj` using any of the input arguments in the preceding syntaxes.

## Examples

### Simulate a SimBiology Model

Load a sample SimBiology model.

```
sbioloadproject radiodecay.sbproj
```

Change the simulation stop time to 15 seconds.

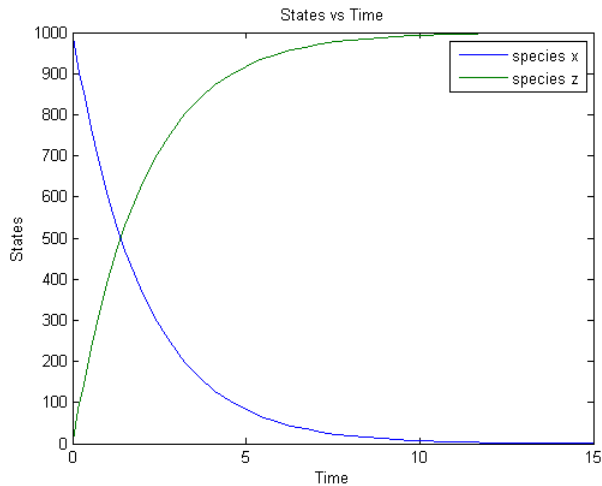
```
csObj = getConfigset(m1,'active');  
set(csObj,'Stoptime',15);
```

Simulate the model and return outputs in an array.

```
[t,x,n] = sbiosimulate(m1);
```

Plot the simulated results for species x and z.

```
figure;  
plot(t,x)  
xlabel('Time')  
ylabel('States')  
title('States vs Time')  
legend('species x','species z')
```



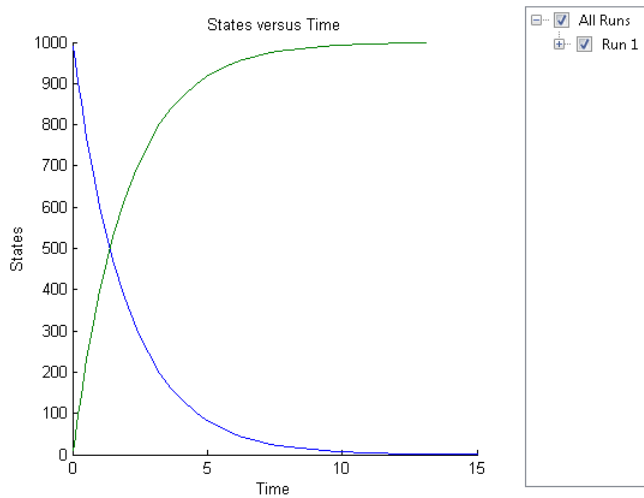
You can also return the results to a SimData object .

```
simData = sbiosimulate(m1);
```

Plot the simulated results.

```
sbioplot(simData);
```





## Simulate a SimBiology Model Using an Array of Dose Objects

Load a sample SimBiology model.

```
sbioloadproject radiodecay.sbproj
```

Add two doses of 100 molecules each for species x, scheduled at 2 and 4 seconds respectively.

```
dObj1 = adddose(m1, 'd1', 'schedule');
dObj1.Amount = 100;
dObj1.AmountUnits = 'molecule';
dObj1.TimeUnits = 'second';
dObj1.Time = 2;
dObj1.TargetName = 'unnamed.x';
```

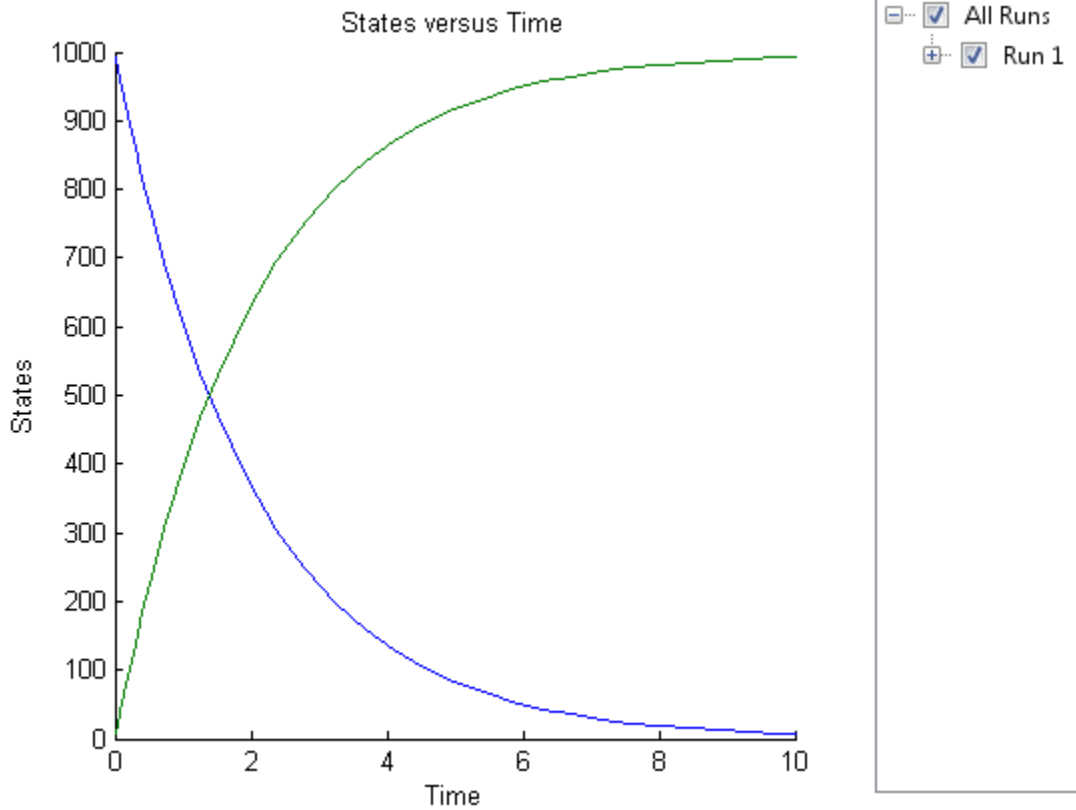
```
dObj2 = adddose(m1, 'd2', 'schedule');
dObj2.Amount = 100;
dObj2.AmountUnits = 'molecule';
dObj2.TimeUnits = 'second';
dObj2.Time = 4;
dObj2.TargetName = 'unnamed.x';
```

Simulate the model using no dose or any subset of the dose array.

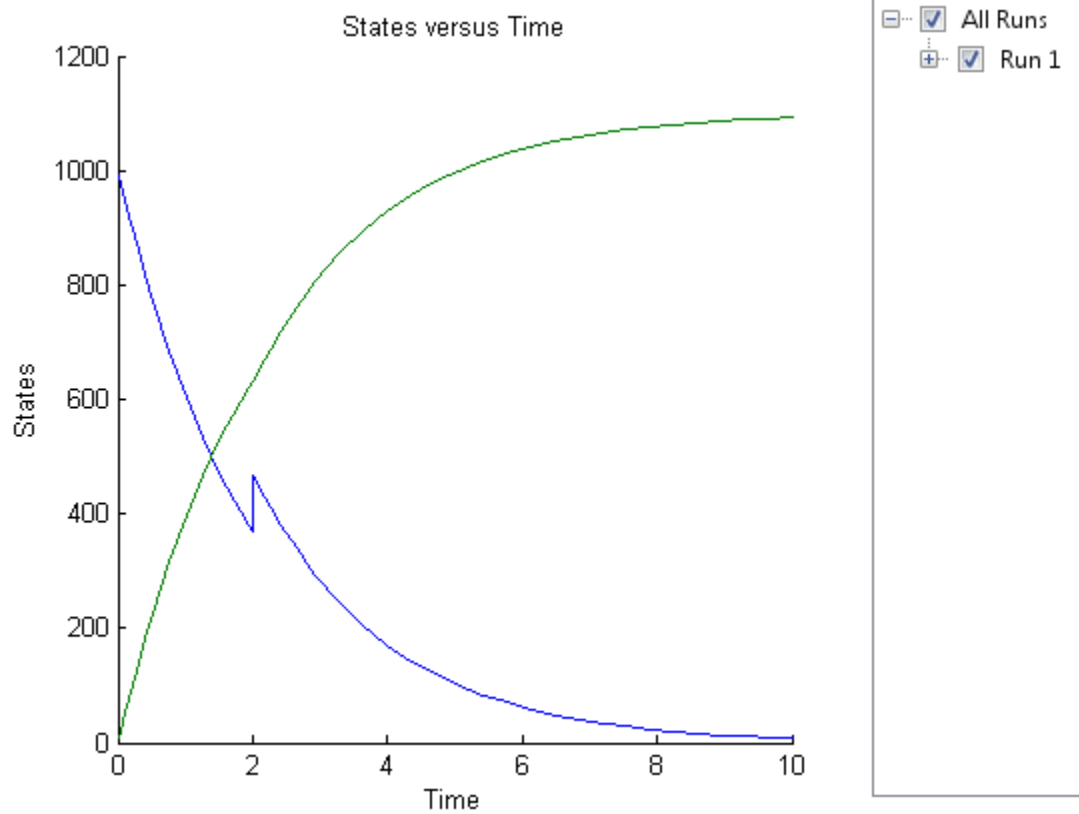
```
sim1 = sbiosimulate(m1);  
sim2 = sbiosimulate(m1,dObj1);  
sim3 = sbiosimulate(m1,dObj2);  
sim4 = sbiosimulate(m1,[dObj1,dObj2]);
```

Plot the results.

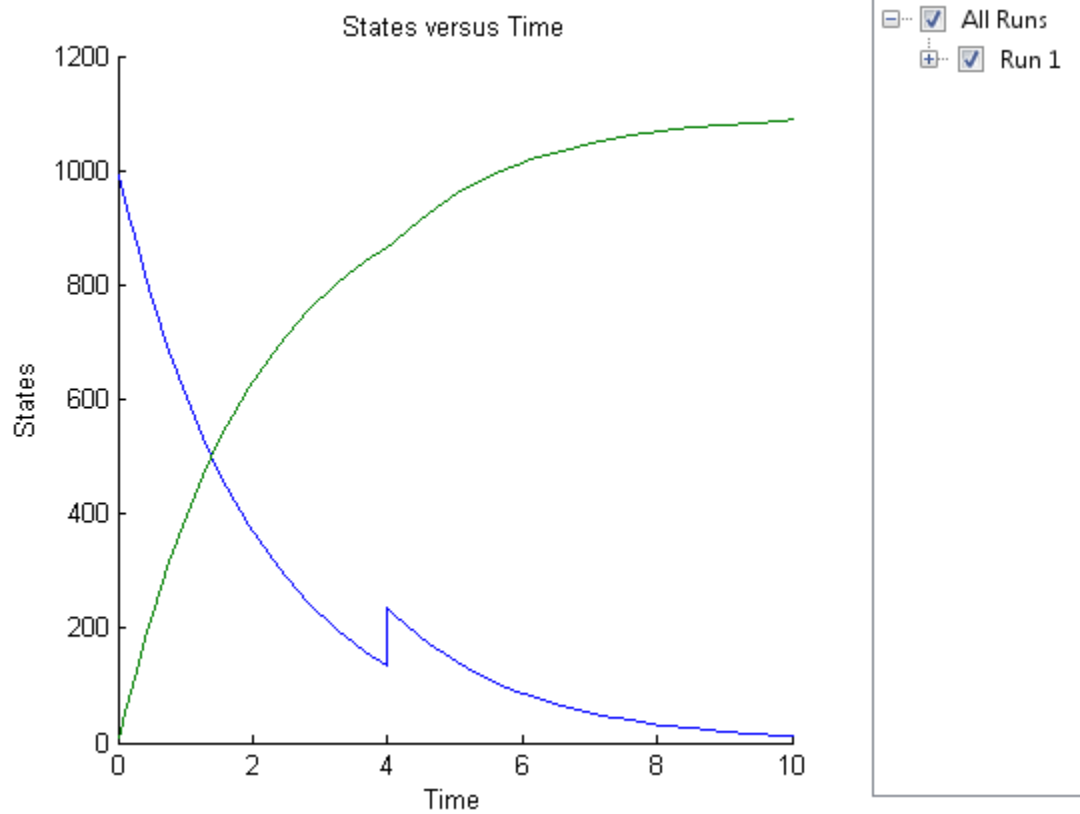
```
sbioplot(sim1)
```



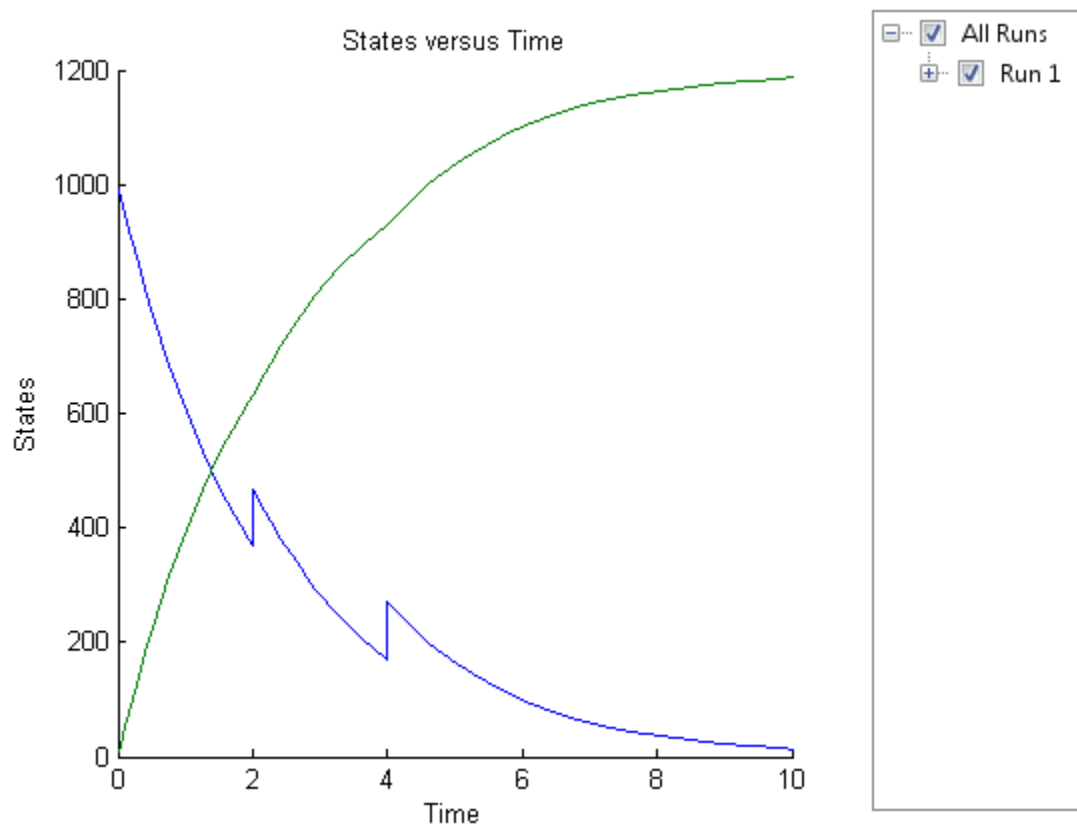
```
sbioplot(sim2)
```



```
sbioplot(sim3)
```



`sbioplot(sim4)`



### Simulate a SimBiology Model Using Configset and Dose Objects

Load a sample SimBiology model.

```
sbioloadproject radiodecay.sbproj
```

Get the default configuration set from the model.

```
defaultConfigSet = getconfigset(m1,'default');
```

Add a scheduled dose of 100 molecules at 2 seconds for species x.

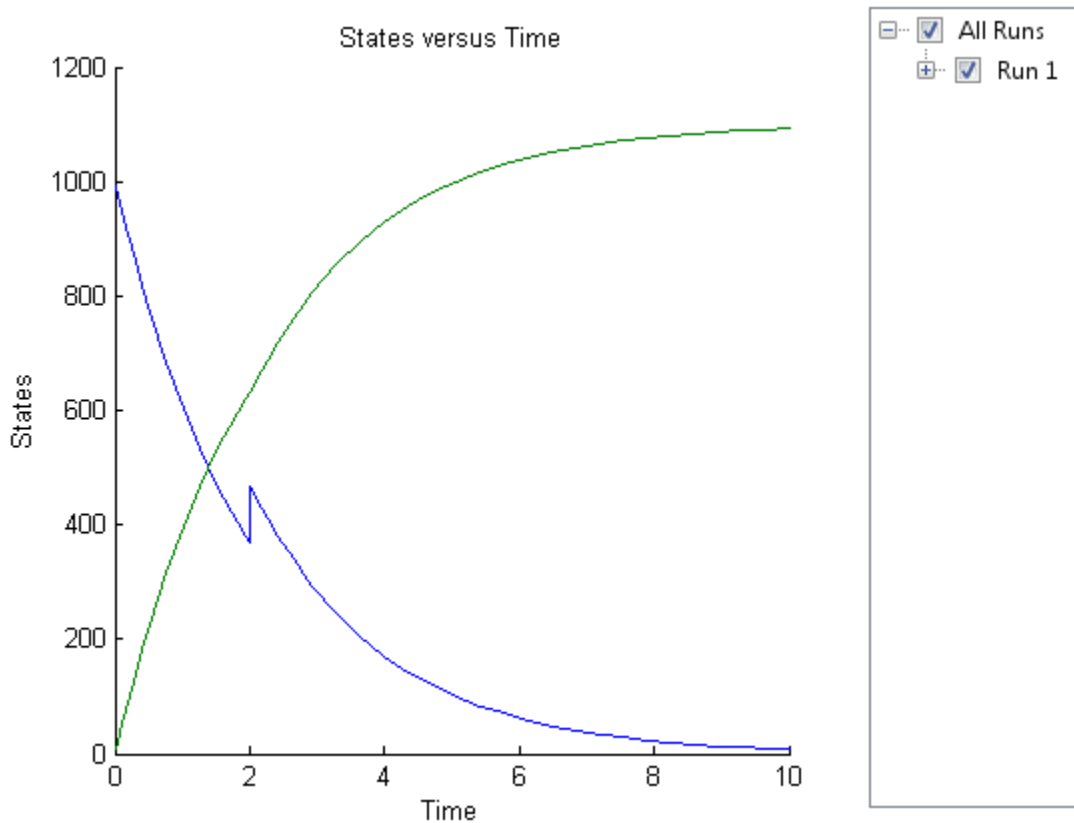
```
dObj = adddose(m1, 'd1', 'schedule');  
dObj.Amount = 100;  
dObj.AmountUnits = 'molecule';  
dObj.TimeUnits = 'second';  
dObj.Time = 2;  
dObj.TargetName = 'unnamed.x';
```

Simulate the model using configset and dose objects.

```
sim = sbiosimulate(m1, defaultConfigSet, dObj);
```

Plot the result.

```
sbioplot(sim);
```



### Simulate a SimBiology Model Using Configset, Dose, and Variant Objects

Load a sample SimBiology model.

```
sbioimportproject radiodecay.sbproj
```

Add a new configuration set using a stop time of 15 seconds.

```
csObj = m1.addconfigset('newStopTimeConfigSet');  
csObj.StopTime = 15;
```

Add a scheduled dose of 100 molecules at 2 seconds for species x.

```
dObj = adddose(m1, 'd1', 'schedule');  
dObj.Amount = 100;  
dObj.AmountUnits = 'molecule';  
dObj.TimeUnits = 'second';  
dObj.Time = 2;  
dObj.TargetName = 'unnamed.x';
```

Add a variant of species *x* using a different initial amount of 500 molecules.

```
vObj = addvariant(m1, 'v1');  
addcontent(vObj, {'species', 'x', 'InitialAmount', 500});
```

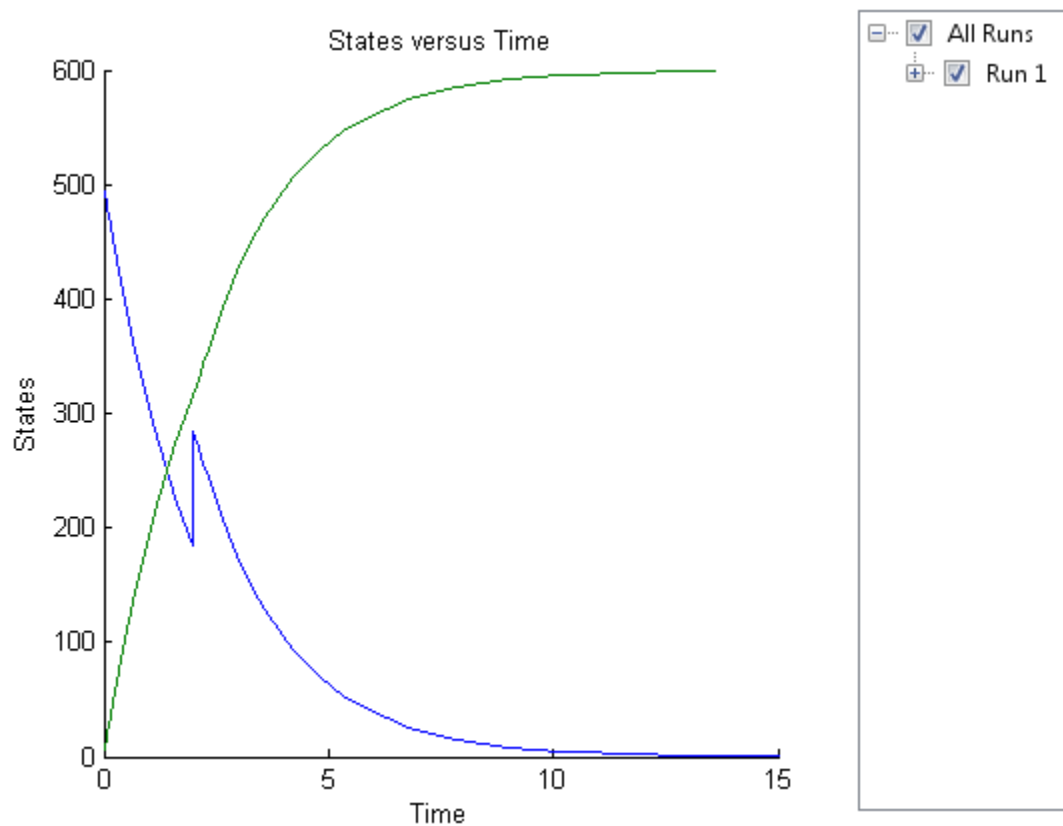
Simulate the model using the same configset, variant, and dose objects. Use the same order of input arguments as shown next.

```
sim = sbiosimulate(m1, csObj, vObj, dObj);
```

Plot the result.

```
sbioplot(sim);
```





## Input Arguments

### **modelObj** — SimBiology model

SimBiology model object

SimBiology model, specified as a SimBiology model object. The model minimally needs one reaction or rate rule for simulations.

### **csObj** — Configuration set object

configset object | []

Configuration set object, specified as a `configset` object that stores simulation-specific information. When you specify `csObj` as `[]`, `sbiosimulate` uses the currently active `configset` object.

If your model contains events, the `csObj` object cannot specify `'expltau'` or `'impltau'` for the `SolverType` property.

If your model contains doses, the `csObj` object cannot specify `'ssa'`, `'expltau'`, or `'impltau'` for the `SolverType` property.

### **dvObj — Dose or variant object**

dose object or array of dose objects | variant object or array of variant objects | `[]`

Dose or variant object, specified as a `ScheduleDose` object, `RepeatDose` object, an array of dose objects, `variant` object, or an array of variant objects.

- Use `[]` when you want to explicitly exclude any variant objects from the `sbiosimulate` function.
- When `dvObj` is a dose object, `sbiosimulate` uses the specified dose object as well as any active variant objects if available.
- When `dvObj` is a variant object, `sbiosimulate` uses the specified variant object as well as any active dose objects if available.

### **variantObj — Variant object**

variant object or array of variant objects | `[]`

Variant object, specified as a `variant` object or an array of variant objects. Use `[]` when you want to explicitly exclude any variant objects from `sbiosimulate`.

### **doseObj — Dose object**

dose object or array of dose objects | `[]`

Dose object, specified as a `ScheduleDose` object, `RepeatDose` object, or an array of dose objects. A dose object defines additions that are made to species amounts or parameter values. Use `[]` when you want to explicitly exclude any dose objects from `sbiosimulate`.

## Output Arguments

### **time** — Vector of time samples

vector

Vector of time samples, returned as an  $n$ -by-1 vector containing the simulation time steps.  $n$  is the number of time samples.

### **x** — Simulation data

array

Simulation data, returned as an  $n$ -by- $m$  data array, where  $n$  is the number of time samples and  $m$  is the number of states logged in the simulation. Each column of  $x$  describes the variation in the quantity of a species, compartment, or parameter over time.

### **names** — Names of species, compartments, or parameters

cell array of character vectors

Names of species, compartments, or parameters, returned as an  $m$ -by-1 cell array of character vectors. In other words, `names` contains the column labels of the simulation data,  $x$ . If the species are in multiple compartments, species names are qualified with the compartment name in the form `compartmentName.speciesName`.

### **simDataObj** — Simulation data

`SimData` object

Simulation data, returned as a `SimData` object that holds time and state data as well as metadata, such as the types and names for the logged states or the configuration set used during simulation. You can access time, data, and names stored in a `SimData` object by using its properties.

## See Also

`Configset` object | `Model` object | `RepeatDose` object | `ScheduleDose` object | `SimData` object | `Variant` object | `addconfigset` | `getconfigset` | `sbioaccelerate` | `sbiomodel` | `setactiveconfigset`

## Topics

“Model Simulation”

“Deriving ODEs from Reactions”

**Introduced before R2006a**

# sbiosteadystate

Find steady state of SimBiology model

## Syntax

```
[success, variant_out] = sbiosteadystate(model)
[success, variant_out] = sbiosteadystate(model, variant_in)
[success, variant_out] = sbiosteadystate(model, variant_in,
scheduleDose)
[success, variant_out, model_out] = sbiosteadystate(model, ___)
[success, variant_out, model_out, exitInfo] = sbiosteadystate(model,
___)
[___] = sbiosteadystate(___ , Name,Value)
```

## Description

[success, variant\_out] = sbiosteadystate(model) attempts to find a steady state of a SimBiology model, model. The function returns success, which is true if a steady state was found, and a SimBiology variant object, variant\_out, with all non-constant species, compartments, and parameters of the model having the steady-state values. If a steady state was not found, then the success is false and variant\_out contains the last values found by the algorithm.

[success, variant\_out] = sbiosteadystate(model, variant\_in) applies the alternate quantity values stored in a SimBiology variant object, variant\_in, to the model before trying to find the steady-state values.

[success, variant\_out] = sbiosteadystate(model, variant\_in, scheduleDose) applies a SimBiology schedule dose object on page 2-433, scheduleDose, or a vector of schedule doses to the corresponding model quantities before trying to find the steady state values. Only doses at time = 0 are allowed, that is, the dose time of each dose object must be 0. To specify a dose without specifying a variant, set variant\_in to an empty array, [].

[success, variant\_out, model\_out] = sbiosteadystate(model, \_\_\_) also returns a SimBiology model, model\_out that is a copy of the input model with the states

set to the steady-state solution that was found. Also, `model_out` has all initial assignment rules disabled.

```
[success, variant_out, model_out, exitInfo] = sbiosteadystate(model, ___)
```

 also returns the exit information about the steady state computation.

```
[ ___ ] = sbiosteadystate( ___ , Name, Value)
```

 uses additional options specified by one or more `Name, Value` pair arguments.

## Examples

### Find a steady state of a simple gene regulation model

This example shows how to find a steady state of a simple gene regulation model, where the protein product from translation controls transcription.

Load the sample SimBiology project containing the model, `m1`. The model has five reactions and four species.

```
sbioloadproject('gene_reg.sbproj', 'm1')
```

Display the model reactions.

```
m1.Reactions
```

```
SimBiology Reaction Array
```

Index:	Reaction:
1	DNA -> DNA + mRNA
2	mRNA -> mRNA + protein
3	DNA + protein <-> DNA_protein
4	mRNA -> null
5	protein -> null

A steady state calculation attempts to find the steady state values of non-constant quantities. To find out which model quantities are non-constant in this model, use `sbioselect`.

```
sbioselect(m1, 'Where', 'Constant*', '==', false)
```

```
SimBiology Species Array
```

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	DNA	50	molecule
2	unnamed	DNA_protein	0	molecule
3	unnamed	mRNA	0	molecule
4	unnamed	protein	0	molecule

There are four species that are not constant, and the initial amounts of three of them are set to zero.

Use `sbiosteadystate` to find the steady state values for those non-constant species.

```
[success,variantOut] = sbiosteadystate(m1)
```

```
success = logical
        1
```

```
SimBiology Variant - SteadyState (inactive)
```

ContentIndex:	Type:	Name:	Property:	Value:
1	compartment	unnamed	Capacity	1
2	species	DNA	InitialAmount	8.79024
3	species	DNA_protein	InitialAmount	41.2098
4	species	mRNA	InitialAmount	1.17203
5	species	protein	InitialAmount	23.4406
6	parameter	Transcription.k1	Value	0.2
7	parameter	Translation.k2	Value	20
8	parameter	[Binding/Unbin...	Value	0.2
9	parameter	[Binding/Unbin...	Value	1
10	parameter	[mRNA Degradat...	Value	1.5
11	parameter	[Protein Degra...	Value	1

The initial amounts of all species of the model have been set to the steady-state values. DNA is a conserved species since the total of DNA and DNA\_protein is equal to 50.

You can also use a variant to store alternate initial amounts and use them during the steady state calculation. For instance, you could set the initial amount of DNA to 100 molecules instead of 50.

```
variantIn = sbiovariant('v1');
addcontent(variantIn,{'species','DNA','InitialAmount',100});
[success2,variantOut2,m2] = sbiosteadystate(m1,variantIn)
```

```
success2 = logical
1
```

SimBiology Variant - SteadyState (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	compartment	unnamed	Capacity	1
2	species	DNA	InitialAmount	12.7876
3	species	DNA_protein	InitialAmount	87.2124
4	species	mRNA	InitialAmount	1.70502
5	species	protein	InitialAmount	34.1003
6	parameter	Transcription.k1	Value	0.2
7	parameter	Translation.k2	Value	20
8	parameter	[Binding/Unbin...	Value	0.2
9	parameter	[Binding/Unbin...	Value	1
10	parameter	[mRNA Degradat...	Value	1.5
11	parameter	[Protein Degra...	Value	1

SimBiology Model - cell

Model Components:

Compartments:	1
Events:	0
Parameters:	6
Reactions:	5
Rules:	0
Species:	4

Since the algorithm has found a steady state, the third output `m2` is the steady state model, where the values of non-constant quantities have been set to steady state values. In this example, the initial amounts of all four species have been updated to steady state values.

`m2.Species`

SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	DNA	12.7876	molecule
2	unnamed	DNA_protein	87.2124	molecule
3	unnamed	mRNA	1.70502	molecule
4	unnamed	protein	34.1003	molecule



## Input Arguments

### **model** — SimBiology model

SimBiology model object

SimBiology model, specified as a `SimBiology model` object.

### **variant\_in** — SimBiology variant

variant object

SimBiology variant, specified as a `variant` object. The alternate quantity values stored in the variant are applied to the model before finding the steady state.

### **scheduleDose** — Dosing information

SimBiology schedule dose object

Dosing information, specified as a SimBiology schedule dose object on page 2-433. The dose must be bolus, that is, there must be no time lag or administration time for the dose. In other words, its `LagParameterName` and `DurationParameterName` properties must be empty, and the dose time (the `Time` property) must be 0. For details on how to create a bolus dose, see “Creating Doses Programmatically”.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'AbsTol', 1e-6` specifies to use the absolute tolerance value of  $10^{-6}$ .

### **Method** — Method to compute steady state

`'auto'` (default) | `'simulation'` | `'algebraic'`

Method to compute the steady state of `model`, specified as the comma-separated pair consisting of `'Method'` and a character vector `'auto'`, `'simulation'`, or `'algebraic'`. The default (`'auto'`) behavior is to use the `'algebraic'` method first. If that method is unsuccessful, the function uses the `'simulation'` method.

For the simulation method, the function simulates the model and uses finite differencing to detect a steady state. For details, see “Simulation Method” on page 1-310.

For the algebraic method, the function computes a steady state by finding a root of the flux function algebraically. For nonlinear models, this method requires Optimization Toolbox. For details, see “Algebraic Method” on page 1-310.

---

**Note** The steady state returned by the algebraic method is not guaranteed to be the same as the one found by the simulation method. The algebraic method is faster since it involves no simulation, but the simulation method might be able to find a steady state when the algebraic method could not.

---

Example: 'Method', 'algebraic'

### **AbsTol — Absolute tolerance to detect convergence**

1e-8 (default) | positive, real scalar

Absolute tolerance to detect convergence, specified as the comma-separated pair consisting of 'AbsTol' and a positive, real scalar.

When you use the algebraic method, the absolute tolerance is used to specify optimization settings and detect convergence. For details, see “Algebraic Method” on page 1-310.

When you use the simulation method, the absolute tolerance is used to determine convergence when finding a steady state solution by forward integration as follows:

$\left( \left\| \frac{d\bar{S}}{dt} \right\| < AbsTol \right) \text{ or } \left( \left\| \frac{d\bar{S}}{dt} \right\| < RelTol * \|\bar{S}\| \right)$ , where  $\bar{S}$  is a vector of nonconstant species, parameters, and compartments.

### **RelTol — Relative tolerance to detect convergence**

1e-6 (default) | positive, real scalar

Relative tolerance to detect convergence, specified as the comma-separated pair consisting of 'RelTol' and a positive, real scalar. This name-value pair argument is used for the simulation method only. The algorithm converges and reports a steady state if the algorithm finds model states by forward integration, such that

$\left( \left\| \frac{d\bar{S}}{dt} \right\| < AbsTol \right) \text{ or } \left( \left\| \frac{d\bar{S}}{dt} \right\| < RelTol * \|\bar{S}\| \right)$ , where  $\bar{S}$  is a vector of non-constant species, parameters, and compartments.

**MaxStopTime — Maximum amount of simulation time to take before terminating without a steady state**

100000 (default) | positive integer

Maximum amount of simulation time to take before terminating without a steady state, specified as the comma-separated pair consisting of 'MaxStopTime' and a positive integer. This name-value pair argument is used for the `simulation` method only.

**MinStopTime — Minimum amount of simulation time to take before searching for a steady state**

1 (default) | positive integer

Minimum amount of simulation time to take before searching for a steady state, specified as the comma-separated pair consisting of 'MinStopTime' and a positive integer. This name-value pair argument is used for the `simulation` method only.

## Output Arguments

**success — Flag to indicate if a steady state of the model is found**

true | false

Flag to indicate if a steady state of the model is found, returned as `true` or `false`.

**variant\_out — SimBiology variant**

variant object

SimBiology variant, returned as a variant object. The variant includes all species, parameters, and compartments of the model with the non-constant quantities having the steady-state values.

**model\_out — SimBiology model at the steady state**

model object

SimBiology model at the steady state, returned as a model object. `model_out` is a copy of the input `model`, with the non-constant species, parameters, and compartments set to the steady-state values. Also, `model_out` has all initial assignment rules disabled. Simulating the model at steady state requires that initial assignment rules be inactive, since these rules can modify the values in `variant_out`.

---

**Note**

- If you decide to commit the `variant_out` to the input `model` that has initial assignment rules, then `model` is not expected to be at the steady state because the rules perturb the system when you simulate the `model`.
  - `model_out` is at steady state only if simulated without any doses.
- 

### **exitInfo — Exit information about steady state computation**

character vector

Exit information about the steady state computation, returned as a character vector. The information contains different messages for corresponding exit conditions.

- `Steady state found (simulation)` - A steady state is found using the simulation method.
- `Steady state found (algebraic)` - A steady state is found using the algebraic method.
- `Steady state found (unstable)` - An unstable steady state is found using the algebraic method.
- `Steady state found (possibly underdetermined)` - A steady state that is, possibly, not asymptotically stable is found using the algebraic method.
- `No Steady state found` - No steady state is found.
- `Optimization Toolbox (TM) is missing` - The method is set to 'algebraic' for nonlinear models and Optimization Toolbox is missing.

## Definitions

### Simulation Method

`sbiosteadystate` simulates the model until `MaxStopTime` on page 1-0 . During the simulation, the function approximates the gradient using finite differencing (forward difference) over time to detect a steady state.

### Algebraic Method

`sbiosteadystate` tries to find a steady state of the model algebraically by finding a root of the flux function  $v$ . The flux function includes reaction equations, rate rules, and

algebraic equations, that is,  $v(X, P) = 0$ , where  $X$  and  $P$  are nonconstant quantities and parameters of the model. Thereby the mass conservation imposed by the reaction equations is respected.

For nonlinear models, `sbiosteadystate` uses `fmincon` to get an initial guess for the root. The solution found by `fmincon` is then improved by `fsolve`. To detect convergence, `sbiosteadystate` uses the absolute tolerance (`'AbsTol'`). In other words, `OptimalityTolerance`, `FunctionTolerance`, and `StepTolerance` options of the corresponding optimization function are set to the `'AbsTol'` value.

For linear models, `sbiosteadystate` finds the roots of the flux function  $v$  by solving a linear system defined by the reaction and conservation equations. For linear models, there are no rate or algebraic equations.

## See Also

`Model` object | `ScheduleDose` object | `Variant` object | `commit` |  
`sbioaccelerate` | `sbiomodel` | `sbiosimulate` | `sbiovariant`

**Introduced in R2016a**

## sbiosubplot

Plot simulation results in subplots

### Syntax

```
sbiosubplot(sd)
sbiosubplot(sd,fcnHandle,xArgs,yArgs)
sbiosubplot(sd,fcnHandle,xArgs,yArgs,showLegend)
sbiosubplot(sd,fcnHandle,xArgs,yArgs,showLegend,Name,Value)
```

### Description

`sbiosubplot(sd)` plots each simulation run from `sd`, a `SimData` object or array of objects, into its own subplot. The subplot is a time plot of each state in `sd`.

`sbiosubplot(sd,fcnHandle,xArgs,yArgs)` plots simulation results by calling the function handle `fcnHandle` with inputs `sd`, `xArgs`, and `yArgs`. The inputs `xArgs` and `yArgs` must be cell arrays of the names of the states to plot.

`sbiosubplot(sd,fcnHandle,xArgs,yArgs,showLegend)` also specifies whether to show the legend in the plot. If `true`, the function shows `yArgs` as the legend.

`sbiosubplot(sd,fcnHandle,xArgs,yArgs,showLegend,Name,Value)` also uses additional options specified by one or more name-value pair arguments. For example, you can specify the x-label and y-label of the plot.

### Examples

#### Plot Selected States from Simulation Data in Subplots

Plot the prey versus predator data from the stochastically simulated lotka model in separate subplots by using a custom function (`plotXY`).

Load the model. Set the solver type to SSA to perform stochastic simulations, and set the stop time to 3.

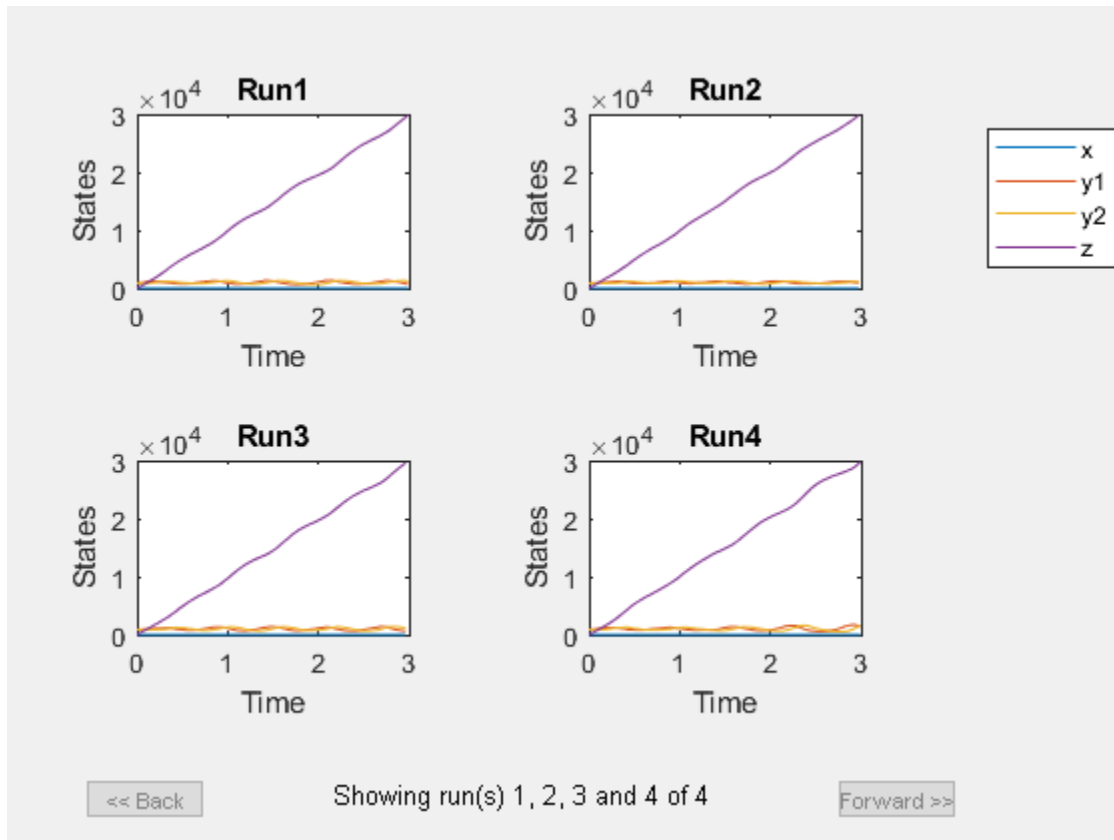
```
sbioloadproject lotka;  
cs              = getconfigset(m1);  
cs.SolverType  = 'SSA';  
cs.StopTime    = 3;  
rng('default') % For reproducibility
```

Set the number of runs and use `sbioensemblerun` for simulation.

```
numRuns = 4;  
sd       = sbioensemblerun(m1,numRuns);
```

Plot each simulation run in a separate subplot. By default, `sbiosubplot` shows the time plot of each species for each run per subplot.

```
sbiosubplot(sd);
```

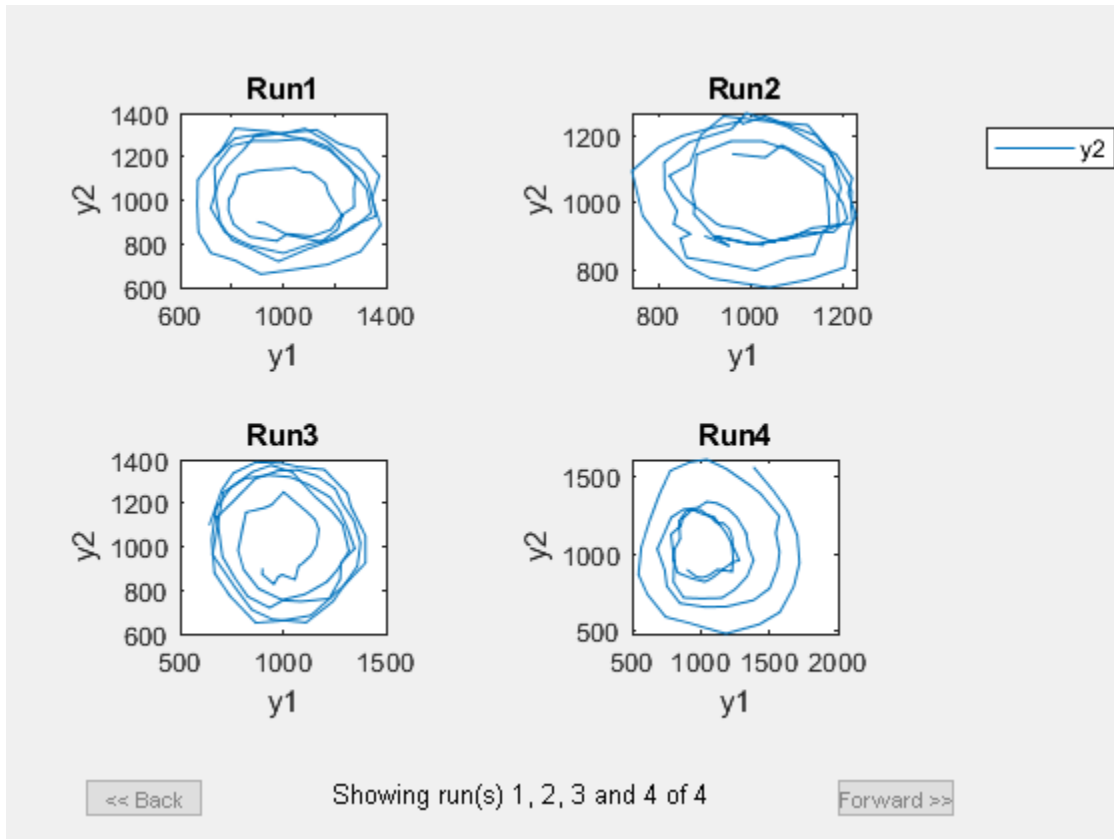


Plot selected states against each other; in this case, plot the prey population versus the predator population in separate subplots for each run. Use the function `plotXY` (shown at the end of this example) to plot the simulated `y1` (prey) data versus the `y2` (predator). Specify the function as a function handle in the `sbiosubplot` call to plot each run in its own subplot. In this case, the fifth input argument (`showLegend`) is set to `true`, which means the fourth input argument (`yArgs`) is shown as the legend.

If you use the live script file for this example, the `plotXY` function is already included at the end of the file. Otherwise, you must define the `plotXY` function at the end of your `.m` or `.mlx` file or add it as a file on the MATLAB path.

```
sbiosubplot(sd,@plotXY,{'y1'},{'y2'},true,'xlabel','y1','ylabel','y2')
```





### Define plotXY Function

sbiosubplot accepts a function handle of a function with the signature:

```
function functionName(sd,xArgs,yArgs).
```

The `plotXY` function plots two selected states against each other. The first input `sd` is the simulation data (SimBiology `SimData` object or vector of objects). In this example, `xArgs` is a cell array containing the name of the species to be plotted on the x-axis, and `yArgs` is a cell array containing the name of the species to be plotted on the y-axis. However, you can use the inputs `xArgs` and `yArgs` in any way in *your* custom plotting function. No output from the function is necessary.

```
function plotXY(sd,xArgs,yArgs)
% Select simulation data for each state from each run.
xDData = selectbyname(sd,xArgs);
yDData = selectbyname(sd,yArgs);
% Plot the species against each other.
plot(xDData.Data,yDData.Data);
end
```

## Input Arguments

### **sd — Simulation results**

SimData object

Simulation results, specified as a SimData object or vector of SimData objects.

This argument corresponds to the first input of the function referenced by fcnHandle.

Example: simdata

### **fcnHandle — Function to generate line plots**

function handle

Function to generate line plots, specified as a function handle. For an example of a custom function to plot selected species from simulation data, see Plot Selected States from Simulation Data in Subplots on page 1-312.

The function must have the signature:

```
function functionName(sd,xArgs,yArgs).
```

The inputs sd, xArgs, and yArgs are the same inputs that you pass in when you call sbiosubplot. No output from the function is necessary.

Example: @plotXY

Data Types: function\_handle

### **xArgs — State names**

string vector | cell array of character vectors

State names to plot, specified as a string vector or cell array of character vectors. For instance, you can use xArgs to represent the states to be plotted on the x-axis of your custom plot.

This argument corresponds to the second input of the function referenced by `fcnHandle`.

Example: `{'y1'}`

Data Types: `cell`

### **yArgs — State names**

string vector | cell array of character vectors

State names to plot, specified as a string vector or cell array of character vectors. For instance, you can use `yArgs` to represent the states to be plotted on the y-axis of your custom plot.

This argument corresponds to the third input of the function referenced by `fcnHandle`.

Example: `{'y2', 'z'}`

Data Types: `cell`

### **showLegend — Logical flag to show plot legend**

`false` (default) | `true`

Logical flag to show the plot legend, specified as `true` or `false`. If `true`, the function shows `yArgs` as the legend.

Example: `true`

Data Types: `logical`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'xlabel', 'Species A'` specifies the x-label of the plot.

### **xlabel — Label for x-axis**

character vector | string

Label for the x-axis of the plot, specified as the comma-separated pair consisting of `'xlabel'` and a character vector or string.

Example: `'xlabel', 'y1'`

Data Types: `char` | `string`

**ylabel** — Label for y-axis

character vector | string

Label for the y-axis of the plot, specified as the comma-separated pair consisting of 'ylabel' and a character vector or string.

Example: 'ylabel', 'y2'

Data Types: `char` | `string`

**See Also**

`sbioplot` | `SimData`

**Introduced in R2008a**

# sbiotrellis

Plot data or simulation results in trellis plot

## Syntax

```
trellisplot = sbiotrellis(data,groupCol,xCol,yCol)
trellisplot = sbiotrellis(data,groupCol,xCol,yCol,Name,Value)
trellisplot = sbiotrellis(data,fcnHandle,groupCol,xCol,yCol)
trellisplot = sbiotrellis(simData,fcnHandle,xCol,yCol)
```

## Description

`trellisplot = sbiotrellis(data,groupCol,xCol,yCol)` plots each group in data as defined by the group column variable `groupCol` into its own subplot. The data defined by column `xCol` is plotted against the data defined by column(s) `yCol`.

`trellisplot = sbiotrellis(data,groupCol,xCol,yCol,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments that are supported by the `plot` command.

`trellisplot = sbiotrellis(data,fcnHandle,groupCol,xCol,yCol)` plots each group in data as defined by the group column variable `groupCol` into its own subplot. `sbiotrellis` creates the subplot by calling the function handle, `fcnHandle`, with input arguments defined by the data columns `xCol` and `yCol`. The `fcnHandle` cannot be empty and must be specified.

The `fcnHandle` must have the signature `fcnHandle(x,y)`, where `x` is a numeric column vector, and `y` is a matrix with the same number of rows as `x`.

For instance, if you want to create a trellis plot with a logarithmic `y`-axis, use `@semilogy` as the function handle, where `semilogy` is the function that plots data with logarithmic scale for the `y`-axis.

`trellisplot = sbiotrellis(simData,fcnHandle,xCol,yCol)` plots each group in `simData` into its own subplot. `sbiotrellis` creates the subplot by calling the function handle, `fcnHandle` with input arguments defined by the columns `xCol` and

`yCol`. The `fcnHandle` can be empty ( `' '` or `[]`). If empty, the default time plot is created using the handle `@plot`.

The `fcnHandle` must have the signature `fcnHandle(simDataI,xCol,yCol)`, where `simDataI` is a single `SimData` object, and `xCol` and `yCol` are the corresponding input arguments to `sbiotrellis`.

---

**Tip** Use the `plot` method of a `sbiotrellis` object to overlay a `SimData` object or a dataset on an existing `sbiotrellis` plot. For example, `plot(trellisplot,...)` adds a plot to the object `trellisplot`. The `SimData` or dataset that is being plotted must have the same number of elements or groups as the `trellisplot` object. The `plot` method has the same input arguments as `sbiotrellis`.

---

## Input Arguments

### **data — Data**

dataset | groupedData object | table

Data, specified as a dataset containing grouped data, a `groupedData` object, or a table.

### **groupCol — Group column name**

character vector | string

Group column name, specified as a character vector or string which is the name of a column in `data` that contains grouping information or an empty name `' '` or `''` which implies there is only one group in `data`.

### **xCol — Name of a column to plot on the x-axis**

character vector | string

Name of a column to plot on the x-axis, specified as a character vector or string.

If `data` is `groupedData`, then `xCol` can also be an empty name `' '` or `''`, and the x-coordinates of the data are determined by the variable specified in `DATA.Properties.IndependentVariableName`.

If `data` is `dataset` or `table`, then `xCol` cannot be empty.

**yCol — Name of a column to plot on the y-axis**

character vector | string | string vector | cell array of character vectors

Name of a column to plot on the y-axis, specified as a character vector, string, string vector, or cell array of character vectors.

**fcnHandle — Handle to a function**

function handle

Handle to a function, specified as a function handle.

If the first argument is a `dataset` or `groupedData` object, the `fcnHandle` must have the signature `fcnHandle(x,y)`, where `x` is a numeric column vector, and `y` is a matrix with the same number of rows as `x`.

If it is a `SimData` object, the `fcnHandle` must have the signature `fcnHandle(simDataI,xCol,yCol)`, where `simDataI` is a single `SimData` object, and `xCol` and `yCol` are the corresponding input arguments to `sbiotrellis`.

**simData — Simulation data**`SimData` object

Simulation data, specified as a `SimData` object.

## Output Arguments

**trellisplot — Plot object**

sbiotrellis object

Plot object, specified as a `sbiotrellis` object. The object has the following properties.

- `hFig` - This is a MATLAB figure object. Use this object to control the appearance and behavior of the figure. For instance, to change the figure window background color to white, enter `trellisplot.hFig.Color = 'white'`. For the list of properties, see the Figure properties.
- `nPlots` - This property tells you the total number of plots in the figure.
- `plots` - This is a vector of axes objects with length equal to `nPlots`. Use this property to control the appearance and behavior of axes objects. For example, if you want to change the y-axis to a log scale, enter

`set(trellisplot.plots, 'YScale', 'log')`. For the list of properties, see the Axes properties.

## Examples

### Create a Trellis Plot for Grouped Data

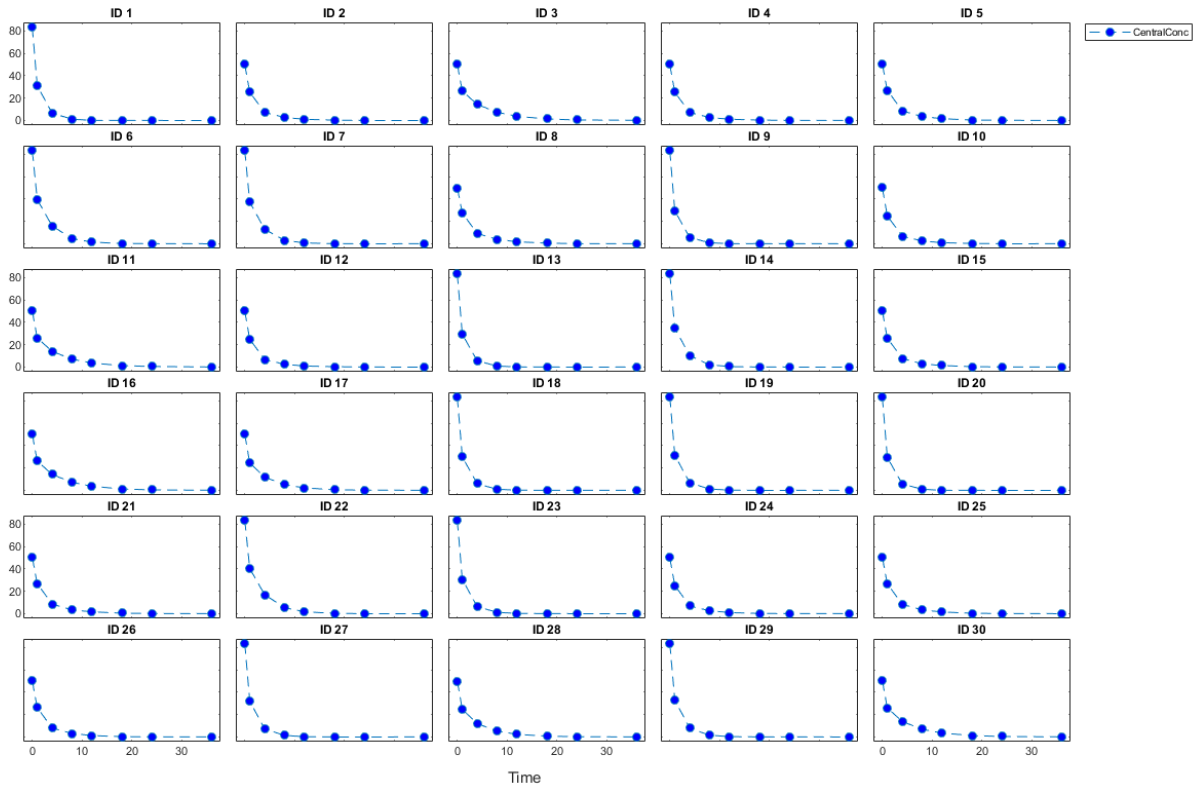
Load a sample dataset. The data contains measurements of drug concentration in the central and peripheral compartments for 30 subjects.

```
load(fullfile(matlabroot, 'examples', 'simbio', 'sd5_302RAgeSex.mat'));
```

Create a trellis plot of the Central concentrations for each subject.

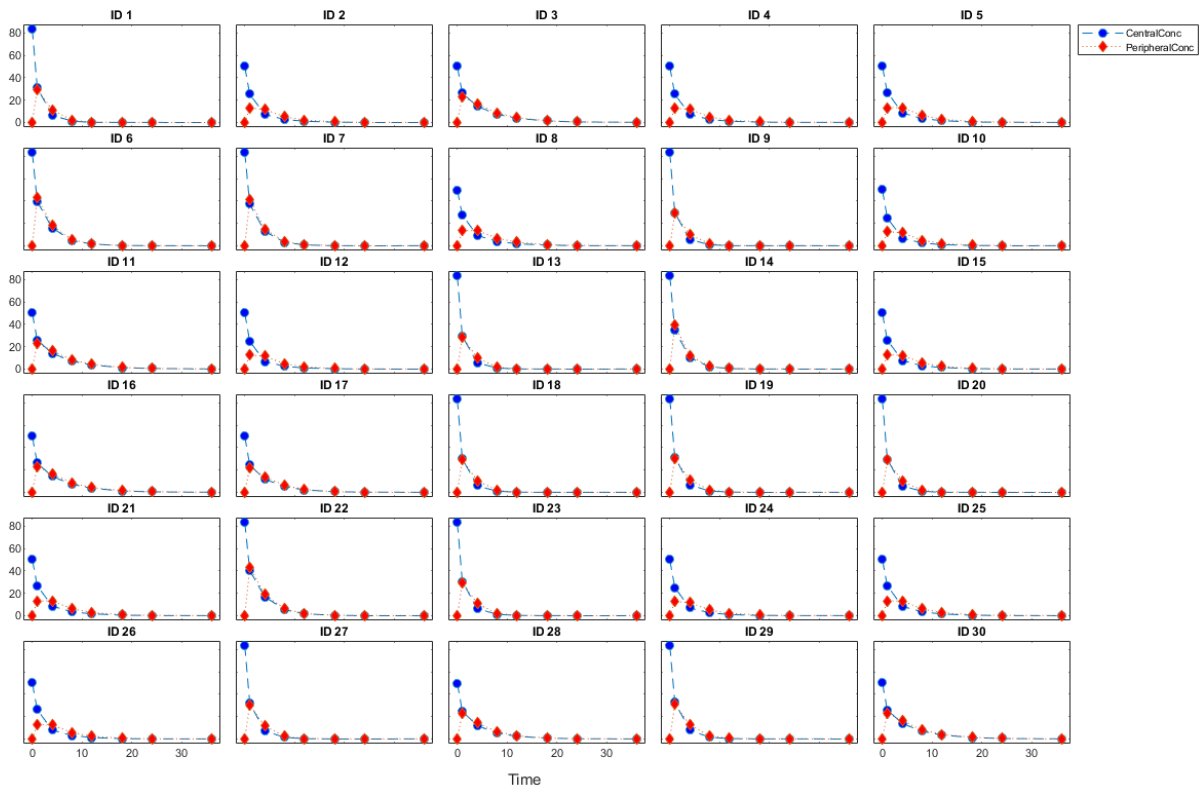
```
t = sbiotrellis(data, 'ID', 'Time', 'CentralConc', ...  
               'Marker', 'o', 'LineStyle', '--', 'MarkerFaceColor', 'b');  
% Resize the figure.  
t.hFig.Position(:) = [100 100 1200 800];
```





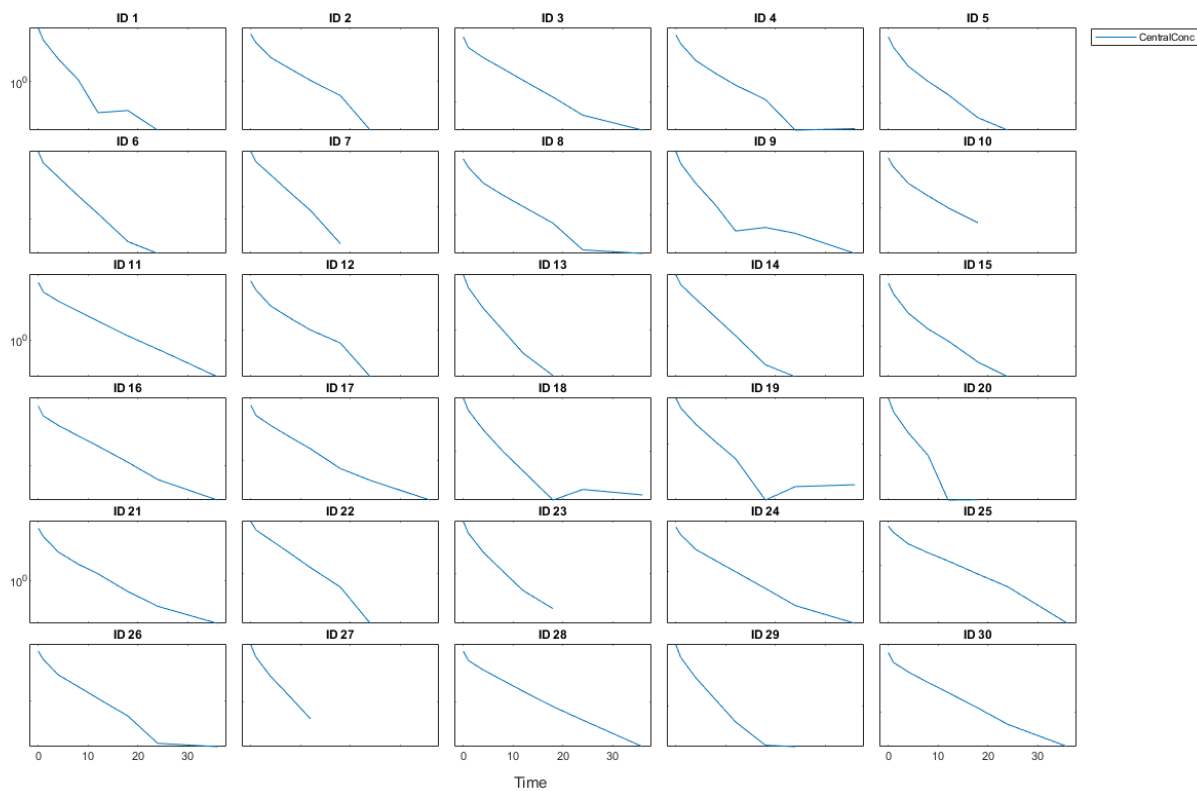
Use the `plot` method of the `sbiotrellis` object to overlay the peripheral concentration data on the same plot.

```
plot(t,data,'ID','Time','PeripheralConc','Marker','d',...  
      'LineStyle',':','MarkerFaceColor','r');
```

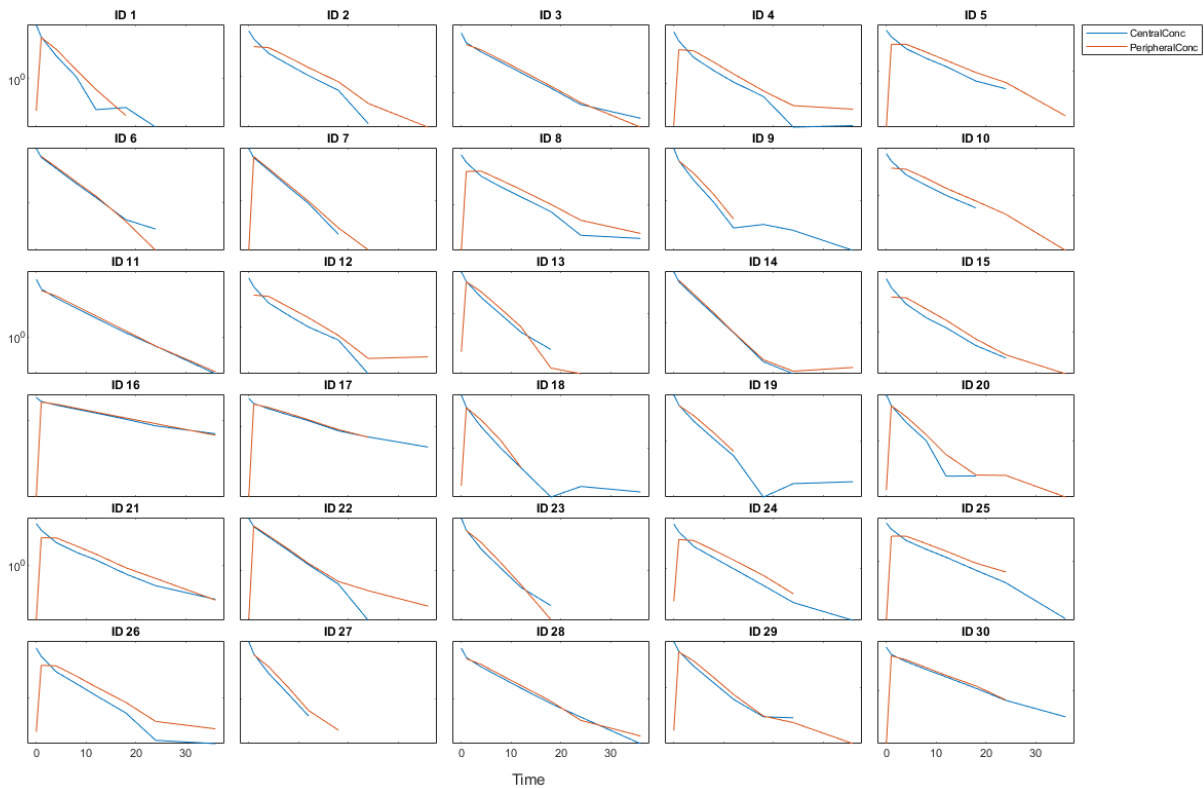


Specify the function handle `@semilogy` to change the y-axis to log scale.

```
t2 = sbiotrellis(data,@semilogy,'ID','Time','CentralConc');
%Resize the figure
t2.hFig.Position(:) = [100 100 1200 800];
```



```
plot(t2,data,@semilogy,'ID','Time','PeripheralConc');
```



## See Also

`sbiplot` | `sbiosubplot`

Introduced in R2009a

# sbiounit

Create user-defined unit

## Syntax

```
unitObject = sbiounit('NameValue')
unitObject = sbiounit('NameValue', 'CompositionValue')
unitObject =
sbiounit('NameValue', 'CompositionValue', MultiplierValue)
unitObject =
sbiounit('NameValue', 'CompositionValue', MultiplierValue, OffsetValue)
unitObject =
sbiounit('NameValue', 'CompositionValue', ... 'PropertyName',
PropertyValue...)
```

## Arguments

<i>NameValue</i>	Name of the user-defined unit. <i>NameValue</i> must begin with characters and can contain characters, underscores, or numbers. <i>NameValue</i> can be any valid MATLAB variable name.
<i>CompositionValue</i>	Shows the combination of base and derived units that defines the unit <i>NameValue</i> . For example <i>molarity</i> is <i>mole/liter</i> . Base units are the set of units used to define all unit quantity equations. Derived units are defined using base units or mixtures of base and derived units.
<i>MultiplierValue</i>	Numerical value that defines the relationship between the user-defined unit <i>NameValue</i> and the base unit as a product of the <i>MultiplierValue</i> and the base unit. For example, 1 mole is $6.0221e23$ *molecule. The <i>MultiplierValue</i> is $6.0221e23$ .

<i>OffsetValue</i>	Numerical value by which the unit composition is modified. For example, °Celsius = (5/9)*(°Fahrenheit-32); °Fahrenheit is <i>Composition</i> ; <i>MultiplierValue</i> is 5/9 and <i>OffsetValue</i> is -32.
<i>PropertyName</i>	Name of the unit object property, for example, 'Notes'.
<i>PropertyValue</i>	Value of the unit object property, for example, 'New unit for GPCR model'.

## Description

*unitObject* = *sbiounit*('NameValue') constructs a SimBiology unit object named *NameValue*. Valid names must begin with a letter, and be followed by letters, underscores, or numbers.

*unitObject* = *sbiounit*('NameValue', 'CompositionValue') allows you to specify the name and the composition of the unit.

*unitObject* = *sbiounit*('NameValue', 'CompositionValue', *MultiplierValue*) creates a unit with the name *NameValue* where the unit is defined as *MultiplierValue*\**CompositionValue*.

*unitObject* = *sbiounit*('NameValue', 'CompositionValue', *MultiplierValue*, *OffsetValue*) creates a unit with the specified offset.

*unitObject* = *sbiounit*('NameValue', 'CompositionValue', ... 'PropertyName', *PropertyValue*...) defines optional properties. The name-value pairs can be in any format supported by the function *set*.

In order to use *unitObject*, you must add it to the user-defined library with the *sbioaddtolibrary* function. To get the unit object into the user-defined library, use the following command:

```
sbioaddtolibrary(unitObject);
```

You can view additional *unitObject* properties with the *get* command. You can modify additional properties with the *set* command. For more information about unit object properties and methods, see Unit object.

Use the `sbiowhos` function to list the units available in the user-defined library.

## Examples

This example shows you how to create a user-defined unit, add it to the user-defined library, and query the library.

- 1 Create units for the rate constants of a first-order and a second-order reaction.

```
unitObj1 = sbiunit('firstconstant', '1/second', 1);
unitObj2 = sbiunit('secondconstant', '1/molarity*second', 1);
```

- 2 Add the unit to the user-defined library.

```
sbioaddtolibrary(unitObj1);
sbioaddtolibrary(unitObj2);
```

- 3 Query the user-defined library in the root object.

```
rootObj = sbioroot;
```

```
rootObj.UserDefinedLibrary.Units
```

```
SimBiology UserDefined Units
```

Index:	Name:	Composition:	Multiplier:	Offset:
1	firstconstant	1/second	1.000000	0.000000
2	secondconstant	1/molarity*second	1.000000	0.000000

Alternatively, use the `sbiowhos` command.

```
sbiowhos -userdefined -unit
```

```
SimBiology UserDefined Units
```

Index:	Name:	Composition:	Multiplier:	Offset:
1	firstconstant	1/second	1.000000	0.000000
2	secondconstant	1/molarity*second	1.000000	0.000000

## See Also

`sbioaddtolibrary` | `sbioshowunits` | `sbiunitprefix` | `sbiowhos`

**Introduced in R2008a**



# sbiunitcalculator

Convert value between units

## Syntax

```
result = sbiunitcalculator('fromUnits', 'toUnits', Value)
```

## Description

`result = sbiunitcalculator('fromUnits', 'toUnits', Value)` converts the value, *Value*, which is defined in the units, *fromUnits*, to the value, *result*, which is defined in the units, *toUnits*.

## Examples

```
result = sbiunitcalculator('mile/hour','meter/second',1)
```

## See Also

`sbioshowunits`

**Introduced in R2006a**

## sbiunitprefix

Create user-defined unit prefix

### Syntax

```
unitprefixObject = sbiunitprefix('NameValue')  
unitprefixObject = sbiunitprefix('NameValue', 'ExponentValue')  
unitprefixObject = sbiunitprefix('NameValue', ... 'PropertyName',  
PropertyValue ...)
```

### Arguments

<i>NameValue</i>	Name of the user-defined unit prefix. <i>NameValue</i> must begin with characters and can contain characters, underscores, or numbers. <i>NameValue</i> can be any valid MATLAB variable name.
<i>ExponentValue</i>	Shows the value of $10^{\text{Exponent}}$ that defines the relationship of the unit <i>Name</i> to the base unit. For example, for the unit picomole, Exponent is $-12$ .
<i>PropertyName</i>	Name of the unit prefix object property. For example, 'Notes'.
<i>PropertyValue</i>	Value of the unit prefix object property. For example, 'New <i>unitprefix</i> for GPCR model'.

### Description

*unitprefixObject* = sbiunitprefix('NameValue') constructs a SimBiology unit prefix object with the name *NameValue*. Valid names must begin with a letter, and be followed by letters, underscores, or numbers.

*unitprefixObject* = sbiunitprefix('NameValue', 'ExponentValue') creates a unit-prefix object with a multiplicative factor of  $10^{\text{ExponentValue}}$ .

`unitprefixObject = sbiunitprefix('NameValue', ...'PropertyName', PropertyValue ...)` defines optional properties. The property name/property value pairs can be in any format supported by the function `set`.

In order to use `unitprefixObject`, you must add it to the user-defined library with the `sbiaddtolibrary` function. To get the unit-prefix object into the user-defined library, use the following command:

```
sbiaddtolibrary(unitprefixObject);
```

You can view additional `unitprefixObject` properties with the `get` command. You can modify additional properties with the `set` command.

Use the `sbishowunitprefixes` function to list the units available in the user-defined library.

## Examples

This example shows how to create a user-defined unit prefix, add it to the user-defined library, and query the library.

- 1 Create a unit prefix.

```
unitprefixObj1 = sbiunitprefix('peta', 15);
```

- 2 Add the unit prefix to the user-defined library.

```
sbiaddtolibrary(unitprefixObj1);
```

- 3 Query the user-defined library in the root object.

```
rootObj = sbioroot;
```

```
rootObj.UserDefinedLibrary.UnitPrefixes
```

```
Unit Prefix Array
```

Index:	Library:	Name:	Exponent:
1	UserDefined	peta	15

Alternatively, use the `sbiwhos` command.

```
sbiwhos -userdefined -unitprefix
```

```
SimBiology UserDefined Unit Prefixes
```

Index:	Name:	Multiplier:
1	peta	1.000000e+015

## See Also

[sbioaddtolibrary](#) | [sbioshowunits](#) | [sbiounit](#) | [sbiowhos](#)

## Topics

[sbioshowunits](#) on page 1-286

**Introduced in R2008a**

# sbiovariant

Construct variant object

## Syntax

```
variantObj = sbiovariant(vName)
variantObj = sbiovariant(vName,vContent)
variantObj = sbiovariant( ____,Name,Value)
```

## Description

`variantObj = sbiovariant(vName)` creates a variant object named `vName`.

`variantObj = sbiovariant(vName,vContent)` creates a variant object and sets its `Content` property to the cell array of data, `vContent`.

`variantObj = sbiovariant( ____,Name,Value)` uses any of the input arguments in the previous syntaxes and additional options specified by one or more `Name,Value` pair arguments. Each name-value pair represents a property and corresponding value of a variant object. To view all the properties for a variant object, use the `get` function.

To append more data to an existing content of a variant, use `addcontent`. To replace existing data in the `Content` property, use the `set` function or dot notation.

## Examples

### Create Variant with Alternate Model Values

Load the G protein model.

```
sbioloadproject gprotein
```

The model already has a variant for a mutant strain.

```
getvariant(m1)
```

SimBiology Variant - mutant (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	kGd	Value	0.004

Create another variant with the kGd value of 0.5, and initial amount of 5000 for the G species.

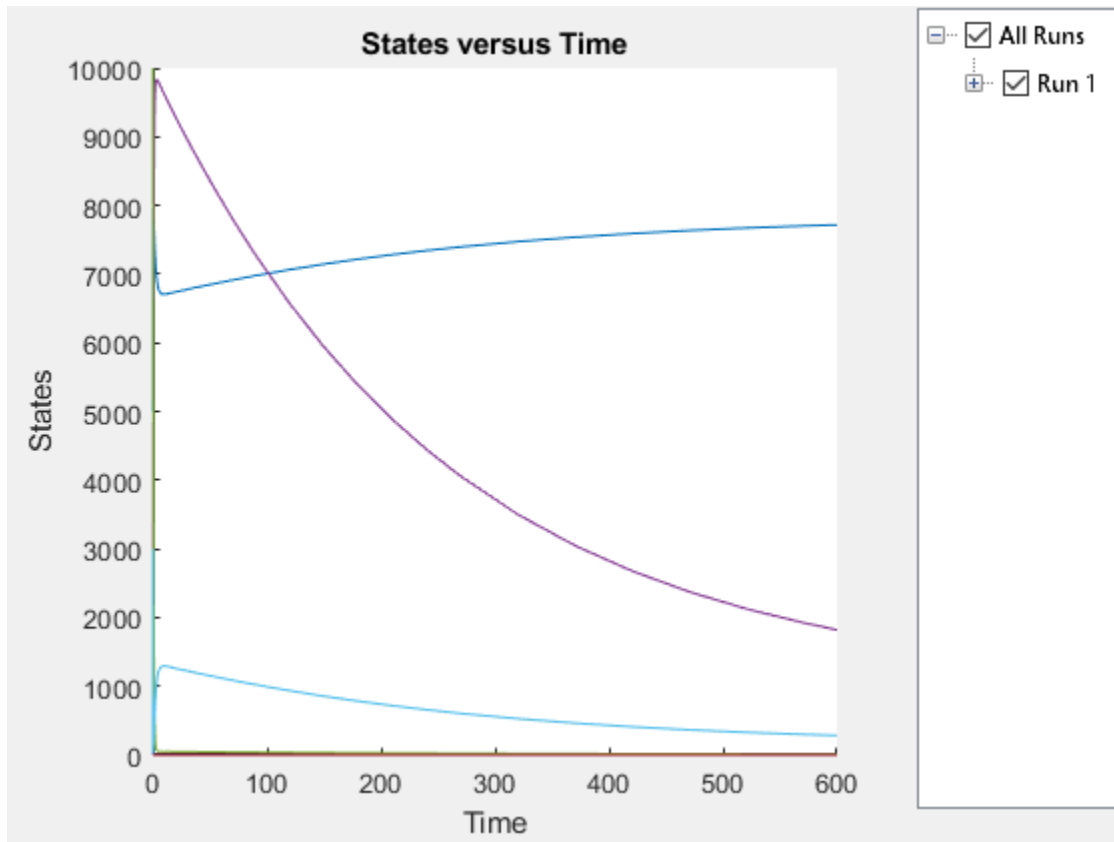
```
v2 = sbiovariant('v2');  
v2.Content = {{'parameter', 'kGd', 'Value', 0.5}, ...  
              {'species', 'G', 'InitialAmount', 5000}}
```

SimBiology Variant - v2 (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	kGd	Value	0.5
2	species	G	InitialAmount	5000

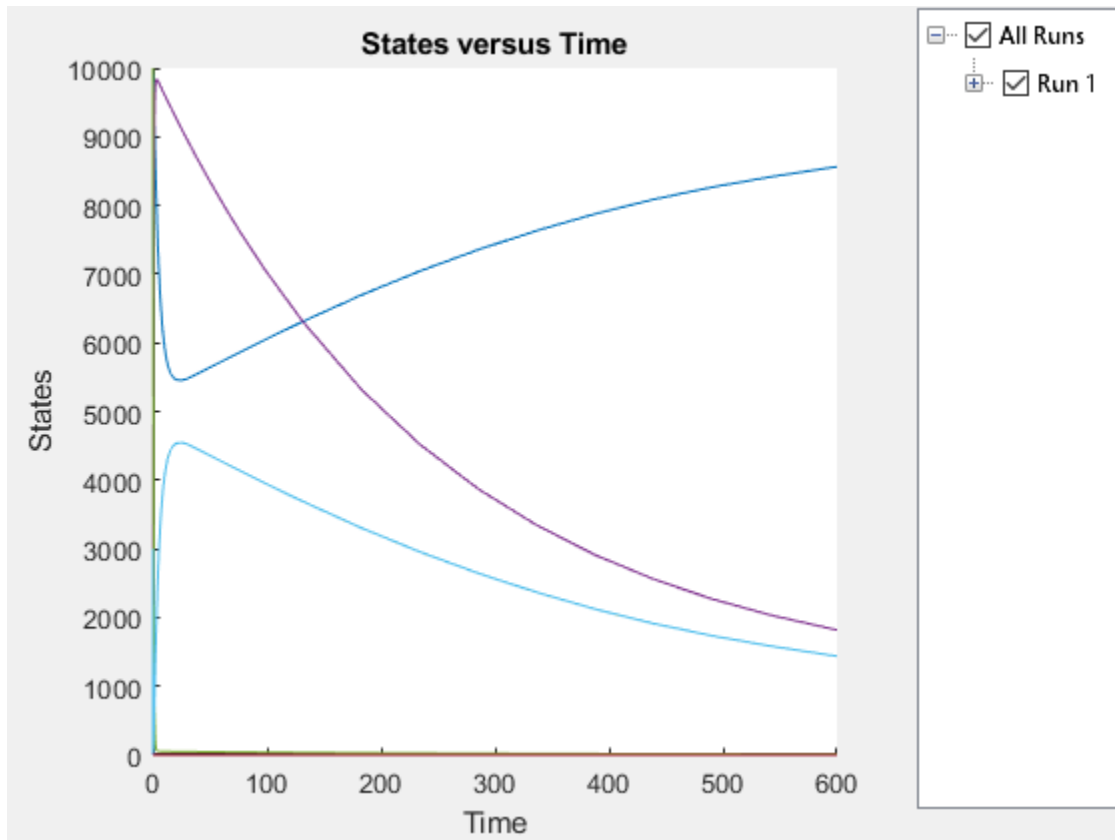
Simulate the model using the initial conditions specified in v2.

```
sbioplot(sbiosimulate(m1,v2));
```



Simulate the wild-type (original) model to compare.

```
sbioplot(sbiosimulate(m1));
```



## Input Arguments

**vName — Name of variant object**

character vector

Name of a variant object, specified as a character vector.

Example: 'cancerPatient'

**vContent — Variant content**

cell array | cell array of cell arrays



Variant content, specified as a cell array or cell array of cell arrays. Each cell array must have four values in this order: the type of model element, its name, its property name, and the alternate value of the property. For instance, `{'species','A','InitialAmount',5}` specifies to store an alternate initial amount of 5 for the species A.

Example: `{'parameter','ka','Value',0.3}`

## Output Arguments

**variantObj** — Variant with alternate model values

variant object

Variant with alternate model values, specified as a `variant` object.

## See Also

`addvariant` | `delete` | `get` | `getvariant` | `set`

## Topics

“Simulate Biological Variability of the Yeast G Protein Cycle Using the Wild-Type and Mutant Strains”

“Simulate Model of Glucose-Insulin Response with Different Initial Conditions”  
“Variants”

**Introduced in R2008a**

## sbiowhos

Show contents of project file, library file, or SimBiology root object

### Syntax

```
sbiowhos flag  
sbiowhos ('flag')  
sbiowhos flag1 flag2...  
sbiowhos FileName
```

### Description

sbiowhos shows contents of the SimBiology root object. This includes the built-in and user-defined kinetic laws, units, and unit prefixes.

sbiowhos flag shows specific information about the SimBiology root object as defined by flag. Valid flags are described in this table.

Flag	Description
-builtin	Built-in kinetic laws, units, and unit prefixes
-data	Data saved in file
-kineticlaw	Built-in and user-defined kinetic laws
-unit	Built-in and user-defined units
-unitprefix	Built-in and user-defined unit prefixes
-userdefined	User-defined kinetic laws, units, and unit prefixes

You can also specify the functional form `sbiowhos ('flag')`.

sbiowhos flag1 flag2... shows information about the SimBiology root object as defined by flag1, flag2,....

sbiowhos FileName shows the contents of the SimBiology project or library defined by Name.

## Examples

- % Show contents of the SimBiology root object  
sbiowhos
  
- % Show kinetic laws on the SimBiology root object  
sbiowhos -kineticlaw
  
- % Show the builtin units of the SimBiology root object.  
sbiowhos -builtin -unit
  
- % Show all contents of project file.  
sbiowhos myprojectfile
  
- % Show kinetic laws from a library file.  
sbiowhos -kineticlaw mylibraryfile
  
- % Show all contents of multiple files.  
sbiowhos myfile1 myfile2

## See Also

whos

**Introduced in R2006a**

## sbmlexport

Export SimBiology model to SBML file

### Syntax

```
sbmlexport(modelObj)  
sbmlexport(modelObj, 'FileName')
```

### Arguments

<i>modelObj</i>	Model object. Enter a variable name for a model object.
<i>FileName</i>	XML file with a Systems Biology Markup Language (SBML) format. Enter either a file name or a path and file name supported by your operating system. If the file name does not have the extension <code>.xml</code> , then <code>.xml</code> is appended to end of the file name.

### Description

`sbmlexport(modelObj)` exports a SimBiology model object (`modelObj`) to a file with a Systems Biology Markup Language (SBML) Level 2 Version 4 format. The default file extension is `.xml` and the file name matches the model name.

`sbmlexport(modelObj, 'FileName')` exports a SimBiology model object (`modelObj`) to an SBML file named *FileName*. The default file extension is `.xml`.

A SimBiology model can also be written to a SimBiology project with the `sbiosaveproject` function to save features not supported by SBML.

For more information about features that are supported by SimBiology but not by SBML or vice visa, see “SBML Support”.

## Examples

Export a model (`modelObj`) to a file (`gene_regulation.xml`) in the current working directory.

```
sbmlexport(modelObj, 'gene_regulation.xml');
```

## References

Finney, A., Hucka, M., (2003), *Systems Biology Markup Language (SBML) Level 2: Structures and facilities for model definitions*. Accessed from SBML.org

## See Also

`sbiomodel` | `sbiosaveproject` | `sbmlimport`

## Topics

“Exporting a SimBiology Model to SBML Format”

“SBML Support”

`sbiomodel` on page 1-149

`sbiosaveproject` on page 1-272

**Introduced before R2006a**

## sbmlimport

Import SBML-formatted file

### Syntax

```
modelObj = sbmlimport(File)
```

### Description

`modelObj = sbmlimport(File)` imports *File*, a Systems Biology Markup Language (SBML)-formatted file, into MATLAB and creates a model object `modelObj`.

*File* is a character vector or string specifying a file name or a path and file name supported by your operating system. *File* extensions are `.sbml` or `.xml`. *File* can also be a URL, if you have the Java<sup>®</sup> programming language.

`sbmlimport` supports SBML Level 3 Version 1 and earlier.

For functional characteristics and limitations, see “SBML Support”.

---

### Note

- `sbmlimport` adds an initial assignment rule when importing SBML models with the following conditions.
  - If an SBML model has a species *s* initialized to *X* using `initialAmount` and has the attribute setting `hasOnlySubstanceUnits = false`:
    - SimBiology sets the initial amount of *s* to *X*.
    - If the model does not already have an initial assignment or repeated assignment rule for *s*, SimBiology adds an initial assignment rule  $s = X / V$ , where *V* is the compartment volume (capacity). This rule ensures that the initial amount of *s* is a concentration unit.

- If the model already has an initial assignment or repeated assignment rule for  $s$ , then SimBiology does not use the value  $X$ . Instead, SimBiology evaluates the rule and sets the appropriate initial amount.
  - If an SBML model has a species  $s$  initialized to  $X$  using `initialConcentration` and has the attribute setting `hasOnlySubstanceUnits = true`:
    - SimBiology sets the initial amount of  $s$  to  $X$ .
    - If the model does not already have an initial assignment or repeated assignment rule for  $s$ , SimBiology adds an initial assignment rule  $s = X * V$ , where  $V$  is the compartment volume (capacity). This rule ensures that the initial amount of  $s$  is an amount unit.
    - If the model already has an initial assignment or repeated assignment rule for  $s$ , then SimBiology does not use the value  $X$ . Instead, SimBiology evaluates the rule and sets the appropriate initial amount.
  - If a species  $s$  in an SBML model has the attribute setting `hasOnlySubstanceUnits = true` without any units defined, SimBiology issues a warning and sets the species amount unit to a default unit (mole) to ensure it is interpreted as an amount, not a concentration. The imported SimBiology model has the `DimensionalAnalysis` property set to `false` to prevent dimensional analysis errors.
- 

## Input Arguments

### File

Character vector or string specifying either of the following:

- File name or path and file name supported by your operating system
- URL (if you have Java programming language)

## Examples

Import SBML model:

```
sbmlobj = sbmlimport('oscillator.xml');
```

## References

Finney, A., Hucka, M., (2003). *Systems Biology Markup Language (SBML) Level 2: Structures and facilities for model definitions*. SBML.org.

## See Also

DimensionalAnalysis | get | sbiosimulate | sbmlexport | set

## Topics

“SBML Support”

“Importing from SBML Files”

**Introduced before R2006a**



# simbiology

Open SimBiology desktop for modeling and simulation

## Syntax

```
simbiology  
simbiology(modelObj)  
simbiology(File)
```

## Input Arguments

<i>modelObj</i>	SimBiology model object or an array of model objects.
<i>File</i>	String specifying a file name or path and file name of an sbproj file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder.

## Description

simbiology opens the “SimBiology Desktop”, which lets you:

- Build a SimBiology model by representing reaction pathways and entering kinetic data for the reactions.
- Import or export SimBiology models to and from the MATLAB workspace or from a Systems Biology Markup Language (SBML) file.
- Modify an existing SimBiology model.
- Simulate a SimBiology model using individual or ensemble runs.
- View results from the simulation.
- Perform analysis tasks such as sensitivity analysis, parameter and species scans, and calculation of conserved moieties.
- Import and plot data for analysis tasks.

- Create and/or modify user-defined units and unit prefixes.
- Create and/or modify user-defined kinetic laws.

`simbiology(modelObj)` opens the SimBiology desktop with `modelObj`, a SimBiology model object. If a project is open in the desktop, the `simbiology` function adds `modelObj` to the project.

`simbiology(File)` opens the project specified by `File` in the SimBiology desktop. `File` is a string specifying a file name or path and file name of an sbproj file. If you specify only a file name, that file must be on the MATLAB search path or in the MATLAB Current Folder. If a project is open in the desktop, the software replaces it with the new project, after prompting you to save any changes.

The `Parent` property of a SimBiology model object is set to the SimBiology root object. The root object contains a list of model objects that are accessible from the MATLAB command line and from the SimBiology desktop. Because both the command line and the desktop point to the same model object in the `Root` object, changes you make to the model at the command line are reflected in the desktop, and vice versa.

---

**Note** The `sbioreset` function removes all models from the root object. Therefore, the `sbioreset` function removes all models from the SimBiology desktop.

---

## Examples

Create a SimBiology model in the MATLAB workspace, and then open the SimBiology desktop with the model:

```
modelObj = sbiomodel('cell');  
simbiology(modelObj)
```

## See Also

`sbiroot`

## Topics

“SimBiology Desktop”

**Introduced in R2009a**

## SimBiology.export.Dose class

### Superclasses:

Exported SimBiology model dose object

### Description

`SimBiology.export.Dose` is the superclass for modifiable export dose objects. An export dose is either of subclass `SimBiology.export.RepeatDose` or `SimBiology.export.ScheduleDose`.

### Construction

Export dose objects are created by the `export` method for SimBiology models. By default, all active doses are export doses, but you can specify which doses to export using the optional `editdoses` input argument to `export`.

`export (model)` Export SimBiology models for deployment and standalone applications

### Properties

#### Amount

Amount of dose, a nonnegative scalar value. This property is read-only if it is parameterized in the export model.

#### AmountUnits

Dose amount units. This property is read-only.

#### DurationParameterName

Parameter specifying length of time to administer a dose. This property is read-only.

**EventMode**

Determine how events that change dose parameters affect in-progress dosing. This property is read-only.

**LagParameterName**

Parameter specifying time lag for the dose. This property is read-only.

**Name**

Name of dose object. This property is read-only.

**Notes**

Text describing dose object. This property is read-only.

**Parent**

Name of the parent export model. This property is read-only.

**Rate**

Rate of dose, a nonnegative scalar value. The default value is 0 (`RepeatDose`) or [] (`ScheduleDose`), that is, the dose is interpreted as a bolus (instantaneous) dose. This property is read-only if it is parameterized in the export model.

**RateUnits**

Units for dose rate. This property is read-only.

**TargetName**

Species receiving dose. This property is read-only.

**TimeUnits**

Time units for dosing. This property is read-only.

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Exported SimBiology Model Dose Objects

Open a sample SimBiology model project, and export the included model object.

```
sbioloadproject('AntibacterialPKPD')  
em = export(m1);
```

Get the editable doses from the exported model object.

```
doses = getdose(em)  
  
doses =  
    1x4 RepeatDose array with properties:
```

```
    Interval  
    RepeatCount  
    StartTime  
    TimeUnits  
    Amount  
    AmountUnits  
    DurationParameterName  
    LagParameterName  
    Name  
    Notes  
    Parent  
    Rate  
    RateUnits  
    TargetName
```

The exported model has 4 repeated dose objects.

Display the 3rd dose object from the exported model object.

```
doses(3)
```

```
ans =  
  RepeatDose with properties:  
  
      Interval: 12  
    RepeatCount: 27  
    StartTime: 0  
    TimeUnits: 'hour'  
      Amount: 500  
    AmountUnits: 'milligram'  
DurationParameterName: 'TDose'  
  LagParameterName: ''  
      Name: '500 mg bid'  
      Notes: ''  
      Parent: 'Antibacterial'  
      Rate: 0  
    RateUnits: ''  
    TargetName: 'Central.Drug'
```

Change the dosing amount for this dose object.

```
doses(3).Amount = 600;
```

```
doses(3)
```

```
ans =  
  RepeatDose with properties:  
  
      Interval: 12  
    RepeatCount: 27  
    StartTime: 0  
    TimeUnits: 'hour'  
      Amount: 600  
    AmountUnits: 'milligram'  
DurationParameterName: 'TDose'  
  LagParameterName: ''  
      Name: '500 mg bid'  
      Notes: ''  
      Parent: 'Antibacterial'  
      Rate: 0  
    RateUnits: ''  
    TargetName: 'Central.Drug'
```

## **See Also**

`SimBiology.export.RepeatDose` | `SimBiology.export.ScheduleDose` | `export`

## **Topics**

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”

“Deploy a SimBiology Model”



# SimBiology.export.ExplicitTauSimulationOptions class

## Superclasses:

Settings for explicit tau-leaping solver of exported SimBiology model

## Description

`SimBiology.export.ExplicitTauSimulationOptions` is the class of simulation options associated with the explicit tau-leaping solver of an export model.

## Construction

Explicit tau simulation options are created by the `export` method for SimBiology models with a stochastic `SolverType` set to `'expltau'`.

`export (model)` Export SimBiology models for deployment and standalone applications

## Properties

### ErrorTolerance

Error tolerance, a scalar value in the range (0,1).

### LogDecimation

Frequency to log stochastic simulation output, a positive integer value.

### MaximumNumberOfLogs

Maximum number of logs criterion to stop simulation, a positive scalar value.

### **MaximumWallClock**

Maximum elapsed wall clock time criterion to stop simulation, a positive scalar value.

### **RandomState**

Random number generator, a positive integer value.

### **SolverType**

Character vector indicating solver type to use for simulation, 'expltau'. This property is read only.

### **StopTime**

Simulation time criterion to stop simulation, a nonnegative scalar value.

### **TimeUnits**

Time units for simulation. This property is read only.

## **Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## **See Also**

`SimBiology.export.ImplicitTauSimulationOptions` |  
`SimBiology.export.ODESimulationOptions` |  
`SimBiology.export.SimulationOptions` |  
`SimBiology.export.StochasticSimulationOptions` | `export`

# SimBiology.export.ImplicitTauSimulationOptions class

## Superclasses:

Settings for implicit tau-leaping stochastic simulation of exported SimBiology model

## Description

`SimBiology.export.ImplicitTauSimulationOptions` is the class of simulation options associated with the implicit tau-leaping solver of an export model.

## Construction

Implicit tau-leaping simulation options are created by the `export` method for SimBiology models with a stochastic `SolverType` set to `'impltau'`.

`export (model)` Export SimBiology models for deployment and standalone applications

## Properties

### ErrorTolerance

Error tolerance, a scalar value in the range (0,1).

### LogDecimation

Frequency to log stochastic simulation output, a positive integer value.

### MaxIterations

Nonlinear solver maximum number of iterations, a positive integer value.

### MaximumNumberOfLogs

Maximum number of logs criterion to stop simulation, a positive scalar value.

### **MaximumWallClock**

Maximum elapsed wall clock time criterion to stop simulation, a positive scalar value.

### **RandomState**

Random number generator, a positive integer value.

### **SolverType**

Character vector indicating solver type to use for simulation, 'impltau'. This property is read only.

### **StopTime**

Simulation time criterion to stop simulation, a nonnegative scalar value.

### **TimeUnits**

Time units for simulation. This property is read only.

## **Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## **See Also**

`SimBiology.export.ExplicitTauSimulationOptions` |  
`SimBiology.export.ODESimulationOptions` |  
`SimBiology.export.SimulationOptions` |  
`SimBiology.export.StochasticSimulationOptions` | `export`

# SimBiology.export.Model class

Exported SimBiology model object

## Description

Exported SimBiology models are limited-access models that can be simulated and accelerated. You can speedup simulation of exported models using Parallel Computing Toolbox, and deploy exported models using MATLAB Compiler™.

By default, all active dose objects, species, parameters, and compartments export with a SimBiology model, and are editable in the exported model object. You can limit which doses, species, parameters, and compartments are editable using additional options during export. Reactions, rules, and events are never editable in an exported model.

## Construction

Use the `export` method to export a SimBiology model.

`export (model)` Export SimBiology models for deployment and standalone applications

## Properties

### DependentFiles

Function files the model depends on. This property is read only.

### ExportNotes

Text with additional information associated with the exported model. This property is read only.

## **ExportTime**

Creation time of the exported model. This property is read only.

## **InitialValues**

Vector of initial values for modifiable species, compartments, and parameters.

## **Name**

Name of the exported model. This property is read only.

## **Notes**

HTML text describing the exported model object. This property is read only.

## **SimulationOptions**

`SimBiology.export.SimulationOptions` object specifying simulation options.

## **ValueInfo**

Array of `SimBiology.export.ValueInfo` objects of modifiable species, parameters, and compartments.

## **Methods**

<code>accelerate</code>	Prepare exported SimBiology model for acceleration
<code>getdose</code>	Return exported SimBiology model dose object
<code>getIndex</code>	Get indices into <code>ValueInfo</code> and <code>InitialValues</code> properties
<code>isAccelerated</code>	Determine whether an exported SimBiology model is accelerated
<code>simulate</code>	Simulate exported SimBiology model

## **Copy Semantics**

Handle. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## Examples

### Export SimBiology Model Object

Load a sample SimBiology model object, and export.

```
modelObj = sbmlimport('lotka');
em = export(modelObj)

em =
  Model with properties:
      Name: 'lotka'
  ExportTime: '27-Aug-2018 14:18:23'
  ExportNotes: ''
```

Display the editable values (compartments, species, and parameters) information.

`em.ValueInfo`

```
ans =
  8x1 ValueInfo array with properties:
      Constant
  InitialValue
      Name
      Parent
  QualifiedName
      Tag
      Type
      Units
```

There are 8 editable values. Display the names of the editable values.

`{em.ValueInfo.Name}`

```
ans = 1x8 cell array
  Columns 1 through 7
      {'unnamed'}      {'x'}      {'y1'}      {'y2'}      {'z'}      {'c1'}      {'c2'}
  Column 8
```

```
{'c3'}
```

Display the exported model simulation options.

```
em.SimulationOptions
```

```
ans =  
ODESimulationOptions with properties:  
  
    AbsoluteTolerance: 1.0000e-06  
    AbsoluteToleranceScaling: 1  
    AbsoluteToleranceStepSize: []  
                MaxStep: []  
    OutputTimes: []  
    RelativeTolerance: 1.0000e-03  
    SolverType: 'ode15s'  
    MaximumNumberOfLogs: Inf  
    MaximumWallClock: Inf  
    StopTime: 10  
    TimeUnits: 'second'
```

## See Also

[SimBiology.export.Dose](#) | [SimBiology.export.SimulationOptions](#) | [SimBiology.export.ValueInfo](#) | [export](#)

## Topics

[“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”](#)

[“Deploy a SimBiology Model”](#)

[Class Attributes \(MATLAB\)](#)

[Property Attributes \(MATLAB\)](#)



# SimBiology.export.ODESimulationOptions class

## Superclasses:

Settings for deterministic, ordinary differential equation simulation of exported SimBiology model

## Description

`SimBiology.export.ODESimulationOptions` is the class of simulation options associated with deterministic, ordinary differential equation (ODE) solvers.

## Construction

Deterministic simulation options are created by the `export` method for SimBiology models with a deterministic `SolverType` (for example, `sundials` or `ode15s`).

`export (model)` Export SimBiology models for deployment and standalone applications

## Properties

### **AbsoluteTolerance**

Absolute error tolerance applied to state value during simulation, a positive scalar value.

### **AbsoluteToleranceScaling**

Control scaling of absolute error tolerance, a logical value.

### **AbsoluteToleranceStepSize**

Initial guess for time step size for scaling of absolute error tolerance, [ ] or a scalar value.

## **MaximumNumberOfLogs**

Maximum number of logs criterion to stop simulation, a positive scalar value.

## **MaximumWallClock**

Maximum elapsed wall clock time criterion to stop simulation, a positive scalar value.

## **MaxStep**

Upper bound on ODE solver step size, [ ] or a positive scalar value.

## **OutputTimes**

Times to log in simulation output, a vector of sorted nonnegative values.

## **RelativeTolerance**

Allowable error tolerance relative to state value during simulation, a scalar value in the range (0,1).

## **SolverType**

Character vector indicating solver type to use for simulation. Possible deterministic solver types are:

- 'sundials'
- 'ode15s'
- 'ode23t'
- 'ode45'

## **StopTime**

Simulation time criterion to stop simulation, a nonnegative scalar value.

## **TimeUnits**

Time units for simulation. This property is read only.

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

[SimBiology.export.SimulationOptions](#) |  
[SimBiology.export.StochasticSimulationOptions](#) | [export](#)

## **SimBiology.export.RepeatDose class**

### **Superclasses:**

Repeated doses for exported SimBiology model

### **Description**

`SimBiology.export.RepeatDose` is the class for export repeat doses.

### **Construction**

Export repeat dose objects are created by the `export` method for SimBiology models. By default, all active repeat doses are export repeat doses, but you can specify which repeat doses to export using the optional `editdoses` input argument to `export`.

`export (model)` Export SimBiology models for deployment and standalone applications

### **Properties**

#### **Amount**

Amount of dose, a nonnegative scalar value or the name (character vector) of a model-scoped parameter. This property is read-only if it is parameterized in the export model.

#### **AmountUnits**

Dose amount units. This property is read-only.

#### **DurationParameterName**

Parameter specifying length of time to administer a dose. This property is read-only.

**EventMode**

Determine how events that change dose parameters affect in-progress dosing. This property is read-only.

**Interval**

Time between doses, a nonnegative scalar value or the name of a model-scoped parameter. This property is read-only if it is parameterized in the export model.

**LagParameterName**

Parameter specifying time lag for the dose. This property is read-only.

**Name**

Name of dose object. This property is read-only.

**Notes**

Text describing dose object. This property is read-only.

**Parent**

Name of the parent export model. This property is read-only.

**Rate**

Rate of dose, a nonnegative scalar value or the name (string or character vector) of a model-scoped parameter. This property is read-only if it is parameterized in the export model. The default value is 0, that is, the dose is interpreted as a bolus (instantaneous) dose.

**RateUnits**

Units for dose rate. This property is read-only.

**RepeatCount**

Dose repetitions, a nonnegative integer value or the name (string or character vector) of a model-scoped parameter. This property is read-only if it is parameterized in the export model.

### StartTime

Start time for initial dose, a nonnegative scalar value or the name (string or character vector) of a model-scoped parameter. This property is read-only if it is parameterized in the export model.

### TargetName

Species receiving dose. This property is read-only.

### TimeUnits

Time units for dosing. This property is read-only.

---

**Note** You cannot change the `Rate` property of `RepeatDose` for exported SimBiology model if *all* of the following conditions are true:

- The `UnitConversion` property of the model is already set to `true`.
- The `Rate` property is empty or set to zero.
- The `RateUnits` is set to empty.

To change the `Rate`, do one of the following:

- Set the `UnitConversion` property of the original model to `false`. Then export the model again.
  - Set the `RateUnits` appropriately.
- 

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

`SimBiology.export.Dose` | `SimBiology.export.ScheduleDose` | `export`

# SimBiology.export.ScheduleDose class

## Superclasses:

Schedule dose for exported SimBiology model

## Description

`SimBiology.export.ScheduleDose` is the class for export schedule doses.

## Construction

Export schedule dose objects are created by the `export` method for SimBiology models. By default, all active schedule doses are export schedule doses, but you can specify which schedule doses to export using the optional `editdoses` input argument to `export`.

`export (model)` Export SimBiology models for deployment and standalone applications

## Properties

### Amount

Amount of dose, a nonnegative scalar value.

### AmountUnits

Dose amount units. This property is read-only.

### DurationParameterName

Parameter specifying length of time to administer a dose. This property is read-only.

### EventMode

Determine how events that change dose parameters affect in-progress dosing. This property is read-only.

## **LagParameterName**

Parameter specifying time lag for the dose. This property is read-only.

## **Name**

Name of dose object. This property is read-only.

## **Notes**

Text describing dose object. This property is read-only.

## **Parent**

Name of the parent export model. This property is read-only.

## **Rate**

Rate of dose, a nonnegative scalar value. Default is `[]`, that is, the dose is interpreted as a bolus (instantaneous) dose. For a schedule dose, you can create a combination of bolus and infusion doses by setting the `rate` property to a vector containing zeros and non-zeros.

## **RateUnits**

Units for dose rate. This property is read-only.

## **TargetName**

Species receiving dose. This property is read-only.

## **Time**

Schedule dose times, a vector of nonnegative values.

## **TimeUnits**

Time units for dosing. This property is read-only.

---

**Note** You cannot change the `Rate` property of `ScheduleDose` for exported SimBiology model if *all* of the following conditions are true:



- The `UnitConversion` property of the model is already set to `true`.
- The `Rate` property is empty or set to zero.
- The `RateUnits` is set to empty.

To change the `Rate`, do one of the following:

- Set the `UnitConversion` property of the original model to `false`. Then export the model again.
  - Set the `RateUnits` appropriately.
- 

## Copy Semantics

`Handle`. To learn how handle classes affect copy operations, see [Copying Objects \(MATLAB\)](#).

## See Also

[SimBiology.export.Dose](#) | [SimBiology.export.RepeatDose](#) | [export](#)

## **SimBiology.export.SimulationOptions class**

Simulation settings for exported SimBiology model

### **Description**

`SimBiology.export.SimulationOptions` is the superclass of simulation options for exported models. Simulation options are either of subclass `SimBiology.export.ODESimulationOptions` for deterministic solvers, or `SimBiology.export.StochasticSimulationOptions` for stochastic solvers.

### **Construction**

Simulation options are created by the `export` method for SimBiology models.

`export (model)` Export SimBiology models for deployment and standalone applications

### **Properties**

#### **MaximumNumberOfLogs**

Maximum number of logs criterion to stop simulation, a positive scalar value.

#### **MaximumWallClock**

Maximum elapsed wall clock time criterion to stop simulation, a positive scalar value.

#### **StopTime**

Simulation time criterion to stop simulation, a nonnegative scalar value.

#### **TimeUnits**

Time units for simulation. This property is read only.

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

`SimBiology.export.ODESimulationOptions` |  
`SimBiology.export.StochasticSimulationOptions` | `export`

# SimBiology.export.StochasticSimulationOptions class

## Superclasses:

Settings for stochastic simulation of exported SimBiology model

## Description

`SimBiology.export.StochasticSimulationOptions` is the superclass of simulation options associated with stochastic solvers. The subclasses of `SimBiology.export.StochasticSimulationOptions` are `SimBiology.export.ExplicitTauSimulationOptions` and `SimBiology.export.ImplicitTauSimulationOptions`.

## Construction

Stochastic simulation options are created by the `export` method for SimBiology models with a stochastic `SolverType` (`ssa`, `expltau`, or `impltau`).

`export (model)` Export SimBiology models for deployment and standalone applications

## Properties

### LogDecimation

Frequency to log stochastic simulation output, a positive integer value.

### MaximumNumberOfLogs

Maximum number of logs criterion to stop simulation, a positive scalar value.

**MaximumWallClock**

Maximum elapsed wall clock time criterion to stop simulation, a positive scalar value.

**RandomState**

Random number generator, a positive integer value.

**SolverType**

Character vector indicating solver type to use for simulation. This property is read only. The stochastic solver type is one of:

- 'ssa'
- 'expltau'
- 'impltau'

**StopTime**

Simulation time criterion to stop simulation, a nonnegative scalar value.

**TimeUnits**

Time units for simulation. This property is read only.

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

[SimBiology.export.ExplicitTauSimulationOptions](#) |  
[SimBiology.export.ImplicitTauSimulationOptions](#) |  
[SimBiology.export.ODESimulationOptions](#) |  
[SimBiology.export.SimulationOptions](#) | [export](#)

## updateInitialAssignments

Update initial assignment rules to remove order dependencies

### Syntax

```
updateInitialAssignments(model)
[tfUpdated, ruleChanges, newParameters, modelBackup] =
updateInitialAssignments(model)
```

### Description

`updateInitialAssignments(model)` updates the active initial assignment rules in a SimBiology model to recover the same simulation results at time = 0, as in R2017a or earlier when the initial assignment rules were evaluated according to the order appeared in the model. As of R2017b, the order in which the initial assignment rules appear in the model has no effect on the simulation results. For details, see “Evaluation Order of Rules”.

`[tfUpdated, ruleChanges, newParameters, modelBackup] = updateInitialAssignments(model)` returns a boolean indicating whether the model is updated, `tfUpdated`, a table of changes made to the rules, `ruleChanges`, a vector of newly added parameters, `newParameters`, and a backup copy of the original model, appending the text " (copy)" to the original model name.

### Examples

#### Remove Rule Order Dependencies in SimBiology Model

Load a sample model.

```
sbioloadproject lotka
```

Show the list of species and their initial amounts.

**m1.Species**

SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	x	1	
2	unnamed	y1	900	
3	unnamed	y2	900	
4	unnamed	z	0	

Add two initial assignment rules that can result in different outcomes depending on the order of the rules that appear in the model.

```
addrule(m1, 'x = z', 'initialAssignment');
addrule(m1, 'z = 100', 'initialAssignment');
```

Display the rules.

**m1.Rules**

SimBiology Rule Array

Index:	RuleType:	Rule:
1	initialAssignment	x = z
2	initialAssignment	z = 100

Remove the rule order dependencies from the model. `tf` is a boolean indicating whether the model was updated, `ruleChanges` is a summary table of changes made to the rules, and `newParas` is a vector of newly added parameter objects. `backup` is the copy of the original (unchanged) model.

```
[tf,ruleChanges,newParas,backup] = updateInitialAssignments(m1)
```

```
tf = logical
     1
```

```
ruleChanges=1x3 table
```

UpdatedRule	OldAssignment	NewAssignment
[1x1 SimBiology.Rule]	"x = z"	"x = z0"

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	z0	0	

SimBiology Model - lotka (copy)

Model Components:

Compartments:	1
Events:	0
Parameters:	3
Reactions:	3
Rules:	2
Species:	4

In order to remove order dependencies, SimBiology updated the initial assignment expression  $x = z$  to  $x = z0$ , where  $z0$  is a newly added parameter.

## Input Arguments

### **model** — SimBiology model

model object (default)

SimBiology model, specified as a model on page 2-262 object.

Example: m1

## Output Arguments

### **tfUpdated** — Whether model is updated

true | false

Whether the model is updated, returned as true or false.

### **ruleChanges** — Table of changes made to rules

table

Table of changes made to initial assignment rules, returned as a table with one row per rule. The table contains the following columns.



Column	Description
UpdatedRule	Vector of the updated rule objects in the model.
OldAssignment	String vector of the original Rule property values in the model.
NewAssignment	String vector of the new Rule property values in the model.

### **newParameters — Newly added parameters**

vector

Newly added parameters, returned as a vector of parameter on page 2-271 objects that are referenced in the updated rules.

### **modelBackup — Backup copy of original model**

model object

Backup copy of the original model, returned as a model object.

## **See Also**

Model on page 2-262 | Parameter on page 2-271

## **Topics**

“Evaluation Order of Rules”

### **Introduced in R2017b**

## SimBiology.export.ValueInfo class

Modifiable species, compartments, or parameters in exported SimBiology model

### Description

`SimBiology.export.ValueInfo` is the class that describes the modifiable value components in a `SimBiology.export.Model`, including species, parameters, and compartments.

### Construction

`ValueInfo` objects are created by the `export` method for SimBiology models. By default, all model species, parameters, and compartments are `ValueInfo` objects, but you can specify which value components to export using the optional `editvals` input argument to `export`.

`export (model)` Export SimBiology models for deployment and standalone applications

### Properties

#### Constant

Display whether value is constant or time-varying. This property is read only.

#### InitialValue

Initial value for the component, a scalar value.

#### Name

Name of species, compartment, or parameter. This property is read only.

**Parent**

Name of parent model, compartment, or reaction. This property is read only.

**QualifiedName**

Qualified name of species, compartment, or parameter. This property is read only.

- For compartments and model-scoped parameters, the qualified name is the same as the name.
- For species, the qualified name is `CompartmentName.SpeciesName`.
- For reaction-scoped parameters, the qualified name is `ReactionName.ParameterName`.

**Tag**

Label for species, compartment, or parameter. This property is read only.

**Type**

Type of value (species, parameter, or compartment). This property is read only.

**Units**

Value units. This property is read only

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## See Also

`SimBiology.export.Model` | `export`



# Methods — Alphabetical List

---

The object that the methods apply to are listed in parenthesis after the method name.

## accelerate(SimFunction)

Prepare SimFunction object for accelerated simulations

### Syntax

accelerate(F)

### Input Arguments

F	SimFunction object created by the createSimFunction method of a SimBiology model.
---	---

### Description

accelerate(F) prepares SimFunction object F for accelerated simulations.

---

**Note** F is automatically accelerated at the first function execution. However, manually accelerate the object if you want it accelerated in your deployment applications.

---

### Examples

#### Simulate SimFunction Object

This example uses the Lotka-Volterra (predator-prey) model described by Gillespie [1].

Load the sample project containing the lotka model.

```
sbioloadproject lotka;
```

Create a SimFunction object f with c1 and c2 as input parameters to be scanned, and y1 and y2 as the output of the function with no dose.

```
f = createSimFunction(m1,{'Reaction1.c1', 'Reaction2.c2'},{'y1', 'y2'}, [])
```

```
f =
```

```
SimFunction
```

```
Parameters:
```

Name	Value	Type
'Reaction1.c1'	10	'parameter'
'Reaction2.c2'	0.01	'parameter'

```
Observables:
```

Name	Type
'y1'	'species'
'y2'	'species'

```
Dosed: None
```

The `SimFunction` object `f` is not set for acceleration at the time of creation. But it will be automatically accelerated when executed.

```
f.isAccelerated
```

```
ans =
```

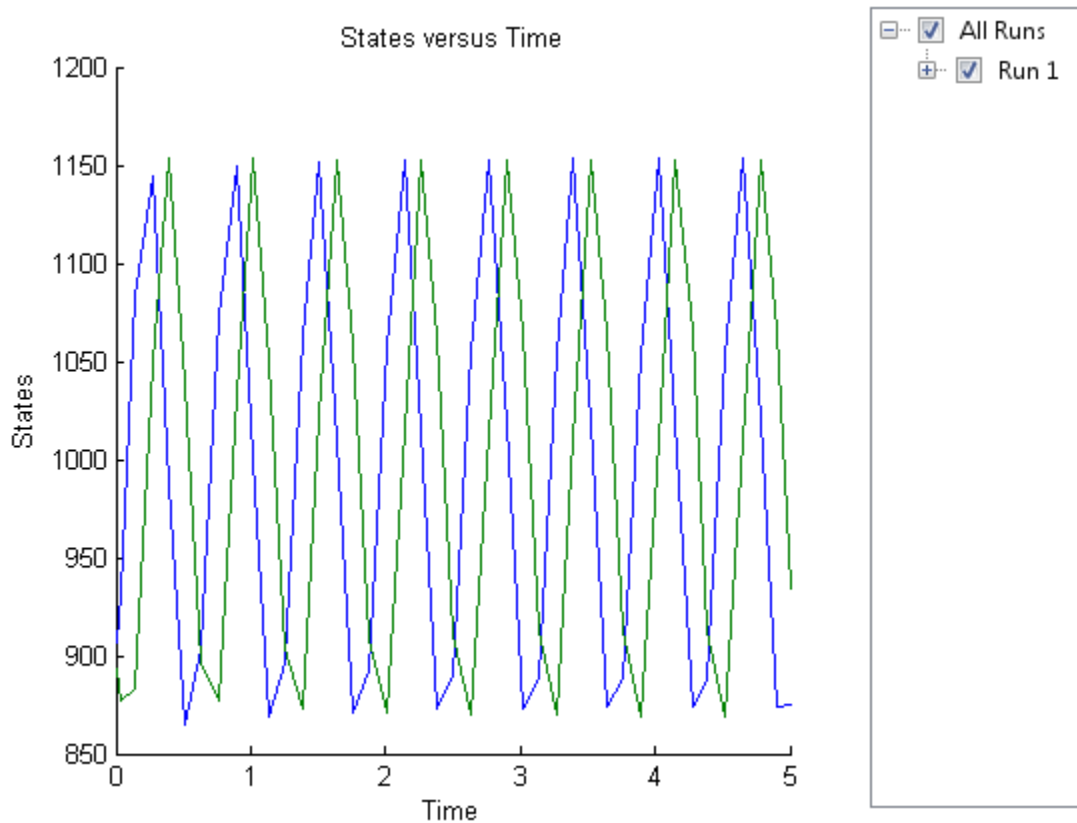
```
0
```

Define an input matrix that contains parameter values for `c1` and `c2`.

```
phi = [10 0.01];
```

Run simulations until the stop time is 5 and plot the simulation results.

```
sbioplot(f(phi,5))
```



Confirm the `SimFunction` object `f` was accelerated during execution.

```
f.isAccelerated
```

```
ans =
```

```
1
```

## See Also

`createSimFunction`, `SimFunction` object



## References

- [1] Gillespie D.T. "Exact Stochastic Simulation of Coupled Chemical Reactions," (1977) The Journal of Physical Chemistry, 81(25), 2340-2361.

**Introduced in R2014a**

## accelerate

**Class:** SimBiology.export.Model

Prepare exported SimBiology model for acceleration

### Syntax

```
accelerate(model)
```

### Description

`accelerate(model)` prepares the exported model for acceleration on the current type of computer.

---

**Note** Microsoft® Visual Studio® 2010 run-time libraries must be available on any computer running accelerated models generated using Microsoft Windows® SDK. If you plan to redistribute your accelerated models to other MATLAB users, be sure they have the run-time libraries.

---

### Input Arguments

**model**

SimBiology.export.Model object.

### Examples

#### Accelerate Exported SimBiology Model

Load a sample SimBiology model object, and export.

```
modelObj = sbmlimport('lotka');
em = export(modelObj)

em =
  Model with properties:
      Name: 'lotka'
      ExportTime: '27-Aug-2018 14:09:30'
      ExportNotes: ''
```

Accelerate the exported model.

```
accelerate(em);
em.isAccelerated
```

```
ans = logical
      1
```

The logical value 1 indicates that the exported model is accelerated.

## See Also

[SimBiology.export.Model](#) | [SimBiology.export.Model.isAccelerated](#) | [export](#)

## Topics

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”  
“Deploy a SimBiology Model”

## AbstractKineticLaw object

Kinetic law information in library

### Description

The abstract kinetic law object represents a *kinetic law definition*, which provides a mechanism for applying a rate law to multiple reactions. The information in this object acts as a mapping template for the reaction rate. The kinetic law definition specifies a mathematical relationship that defines the rate at which reactant species are produced and product species are consumed in the reaction. The expression is shown in the Expression property. The species variables are defined in the SpeciesVariables property, and the parameter variables are defined in the ParameterVariables property of the abstract kinetic law object. For an explanation of how the kinetic law definition relates to the kinetic law object, see KineticLaw object.

Create your own kinetic law definition and add it to the kinetic law library with the `sbioaddtolibrary` function. You can then use the kinetic law to define a reaction rate. To retrieve a kinetic law definition from the user-defined library, first create a root object using `sbioroot`, then use the command `get(rootObj.UserDefinedLibrary, 'KineticLaws')`.

See “Property Summary” on page 2-9 for links to abstract kinetic law object property reference pages.

Properties define the characteristics of an object. For example, an abstract kinetic law object includes properties for the expression, the name of the law, parameter variables, and species variables. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the SimBiology desktop.

### Constructor Summary

`sbioabstractkineticlaw`

Create kinetic law definition

## Method Summary

delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
findUsages (AbstractKineticLaw)	Find out how an AbstractKineticLaw object is used
get (any object)	Get object properties
set (any object)	Set object properties

## Property Summary

Expression (AbstractKineticLaw, KineticLaw)	Expression to determine reaction rate equation
Name	Specify name of object
Notes	HTML text describing SimBiology object
ParameterVariables	Parameters in kinetic law definition
Parent	Indicate parent object
SpeciesVariables	Species in abstract kinetic law
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

Configset object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object, Species object

**Introduced in R2006b**

## addcompartment (model, compartment)

Create compartment object

### Syntax

```
compartmentObj = addcompartment(modelObj, 'NameValue')
```

```
compartmentObj = addcompartment(owningCompObj, 'NameValue')
```

```
compartmentObj = addcompartment(modelObj, 'NameValue',  
CapacityValue)
```

```
compartmentObj = addcompartment(...'PropertyName', PropertyValue...)
```

### Arguments

<i>modelObj</i>	Model object
<i>owningCompObj</i>	Compartment object that contains the newly created compartment object.
<i>NameValue</i>	Name for a compartment object. Enter a character vector unique to the model object.  For information on naming compartments, see Name.
<i>CapacityValue</i>	Capacity value for the compartment object. Enter double. Positive real number, default = 1.
<i>PropertyName</i>	Enter the name of a valid property. Valid property names are listed in “Property Summary” on page 2-12.
<i>PropertyValue</i>	Enter the value for the property specified in <i>PropertyName</i> . Valid property values are listed on each property reference page.

### Description

*compartmentObj* = `addcompartment(modelObj, 'NameValue')` creates a compartment object and returns the compartment object (*compartmentObj*). In the

compartment object, this method assigns a value (*NameValue*) to the property *Name*, and assigns the *model* object (*modelObj*) to the property *Parent*. In the *model* object, this method assigns the compartment object to the property *Compartments*.

*compartmentObj* = addcompartment(*owningCompObj*, '*NameValue*') in addition to the above, adds the newly created compartment within a compartment object (*owningCompObj*), and assigns this compartment object (*owningCompObj*) to the *Owner* property of the newly created compartment object (*compartmentObj*). The parent *model* is the *model* that contains the owning compartment (*owningCompObj*).

*compartmentObj* = addcompartment(*modelObj*, '*NameValue*', *CapacityValue*), in addition to the above, this method assigns capacity (*CapacityValue*) for the compartment.

If you define a reaction within a *model* object (*modelObj*) that does not contain any compartments, the process of adding a reaction generates a default compartment object and assigns the reaction species to the compartment. If there is more than one compartment, you must specify which compartment the species should be assigned to using the format *CompartmentName.SpeciesName*.

*compartmentObj* = addcompartment(...'*PropertyName*', *PropertyValue*...) defines optional properties. "Property Summary" on page 2-12 lists the properties. The *Owner* property is one exception; you cannot set the *Owner* property in the addcompartment syntax because, addcompartment requires the owning *model* or compartment to be specified as the first argument and uses this information to set the *Owner* property.

## Method Summary

Methods for compartment objects

addcompartment (model, compartment)	Create compartment object
addspecies (model, compartment)	Create species object and add to compartment object within model object
copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
findUsages (species, parameter, compartment)	Find out how a species, parameter, or compartment is used in a model
get (any object)	Get object properties
rename (compartment, parameter, species, reaction)	Rename object and update expressions
reorder (model, compartment, kinetic law)	Reorder component lists
set (any object)	Set object properties

## Property Summary

Properties for compartment objects



Capacity	Compartment capacity
CapacityUnits	Compartment capacity units
Compartments	Array of compartments in model or compartment
ConstantCapacity	Specify variable or constant compartment capacity
Name	Specify name of object
Notes	HTML text describing SimBiology object
Owner	Owning compartment
Parent	Indicate parent object
Species	Array of species in compartment object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

### Add Compartments

This example shows how to add compartments to a SimBiology model.

Create a SimBiology model which is named `m1`.

```
model = sbiomodel('m1');
```

Add two compartments to the model, which are named as `Central` and `Peripheral` respectively.

```
comp1 = addcompartment(model, 'Central');  
comp2 = addcompartment(model, 'Peripheral');
```

Change the compartment capacities and units.

```
comp1.Capacity = 2;  
comp1.CapacityUnits = 'liter';  
comp2.Capacity = 1;  
comp2.CapacityUnits = 'liter';
```

Display all the compartments of the model.

### model.Compartments

SimBiology Compartment Array

Index:	Name:	Capacity:	CapacityUnits:
1	Central	2	liter
2	Peripheral	1	liter

## See Also

model object, addproduct, addreactant, addreaction, addspecies

**Introduced in R2007b**

## addCompartment (PKModelDesign)

Add compartment to PKModelDesign object

### Syntax

```
PKCompartmentObj = addCompartment(PKModelDesignObj, CompObjName)
PKCompartmentObj = addCompartment(PKModelDesignObj, CompObjName,
Name, Value)
```

### Description

*PKCompartmentObj* = *addCompartment(PKModelDesignObj, CompObjName)* constructs a PK compartment with the specified name and adds it to *PKModelDesignObj*, a PKModelDesign object.

*PKCompartmentObj* = *addCompartment(PKModelDesignObj, CompObjName, Name, Value)* constructs a PK compartment with the specified name, and with additional options specified by one or more *Name, Value* pair arguments.

### Input Arguments

<i>PKModelDesignObj</i>	PKModelDesign object to which you want to add a compartment
<i>CompObjName</i>	Name of the PKCompartment object that is constructed, specified as a character vector or string.

### Name-Value Pair Arguments

Optional comma-separated pairs of *Name, Value* arguments, where *Name* is the argument name and *Value* is the corresponding value. *Name* must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as *Name1, Value1, ..., NameN, ValueN*.

<p>DosingType</p>	<p>Character vector (or string) specifying the mechanism for drug absorption. Choices are:</p> <ul style="list-style-type: none"> <li>• 'Bolus'</li> <li>• 'Infusion'</li> <li>• 'ZeroOrder'</li> <li>• 'FirstOrder'</li> <li>• '' (default)</li> </ul> <p>For more information, see “Dosing Types”.</p>
<p>EliminationType</p>	<p>Character vector (or string) specifying the mechanism for drug elimination. Choices are:</p> <ul style="list-style-type: none"> <li>• 'Linear'</li> <li>• 'Linear-Clearance'</li> <li>• 'Enzymatic'</li> <li>• '' (default)</li> </ul> <p>For more information, see “Elimination Types”.</p>
<p>HasResponseVariable</p>	<p>Logical indicating if the drug concentration in this compartment is reported. Multiple compartments in a model can have this property set to <code>true</code>. Default is <code>false</code>.</p> <hr/> <p><b>Note</b> If you perform a parameter fit on a model, at least one compartment in the model must have a <code>HasResponseVariable</code> property set to <code>true</code>.</p>
<p>HasLag</p>	<p>Logical indicating if any dose targeting this compartment have a lag associated with them. Default is <code>false</code>.</p>

These optional name-value pair arguments set the corresponding property of the `PKCompartment` object. You can also set these properties after creating the `PKCompartment` object by using the following syntax:

```
PKCompartmentObj.PropertyName = Value
```

For example:

```
PKCompartmentObj.DosingType = 'Bolus'
```

## Output Arguments

<i>PKCompartmentObj</i>	PKCompartment object
-------------------------	----------------------

## Method Summary

get (any object)	Get object properties
set (any object)	Set object properties

## Property Summary

DosingType	Drug dosing type in compartment
EliminationType	Drug elimination type from compartment
HasLag	Lag associated with dose targeting compartment
HasResponseVariable	Compartment drug concentration reported
Name	Specify name of object

## See Also

“Create a Pharmacokinetic Model Using the Command Line”, HasLag, HasResponseVariable, PKCompartment object, PKModelDesign object

**Introduced in R2009a**

## addconfigset (model)

Create configuration set object and add to model object

### Syntax

```
configsetObj = addconfigset(modelObj, 'NameValue')
```

```
configsetObj = addconfigset(..., 'PropertyName', PropertyValue, ...)
```

### Arguments

<i>modelObj</i>	Model object. Enter a variable name.
<i>NameValue</i>	Descriptive name for a configuration set object. Reserved words 'active' and 'default' are not allowed.
<i>configsetObj</i>	Configset object.

### Description

*configsetObj* = addconfigset(*modelObj*, 'NameValue') creates a configuration set object and returns to *configsetObj*.

In the configuration set object, this method assigns a value (*NameValue*) to the property Name.

*configsetObj* = addconfigset(..., 'PropertyName', PropertyValue, ...) constructs a configuration set object, *configsetObj*, and configures *configsetObj* with property value pairs. The *configsetObj* properties are listed in “Property Summary” on page 2-19.

A configuration set stores simulation specific information. A model object can contain multiple configuration sets, with one being active at any given time. The active

configuration set contains the settings that are used during a simulation. *configsetObj* is not automatically set to active. Use the function `setactiveconfigset` to define the active configset for `modelObj`.

Use the method `copyobj` to copy a `configset` object and add it to the `modelObj`.

## Method Summary

Methods for configuration set objects

<code>copyobj</code> (any object)	Copy SimBiology object and its children
<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>set</code> (any object)	Set object properties

## Property Summary

Properties for configuration set objects

Active	Indicate object in use during simulation
AmountUnits	Amount unit used internally during simulation when UnitConversion is on
CompileOptions	Dimensional analysis and unit conversion options
MassUnits	Mass unit used internally during simulation when UnitConversion is on
MaximumNumberOfLogs	Maximum number of logs criteria to stop simulation
MaximumWallClock	Maximum elapsed wall clock time to stop simulation
Name	Specify name of object
Notes	HTML text describing SimBiology object
RuntimeOptions	Options for logged species
SensitivityAnalysisOptions	Specify sensitivity analysis options
SolverOptions	Specify model solver options
SolverType	Select solver type for simulation
StopTime	Simulation time criteria to stop simulation
TimeUnits	Show time units for dosing and simulation
Type	Display SimBiology object type

## Examples

### Add a Configuration Set Object

This example shows how to add a configset object to a SimBiology model and set it up for simulation.

Load the sample radiodecay model `m1`, and add a `Configset` object to the model.

```
sbioloadproject radiodecay;  
configsetObj = addconfigset(m1, 'myset');
```

Configure the simulation stop criteria by setting the `StopTime` property.

```
configsetObj.StopTime = 15;
```

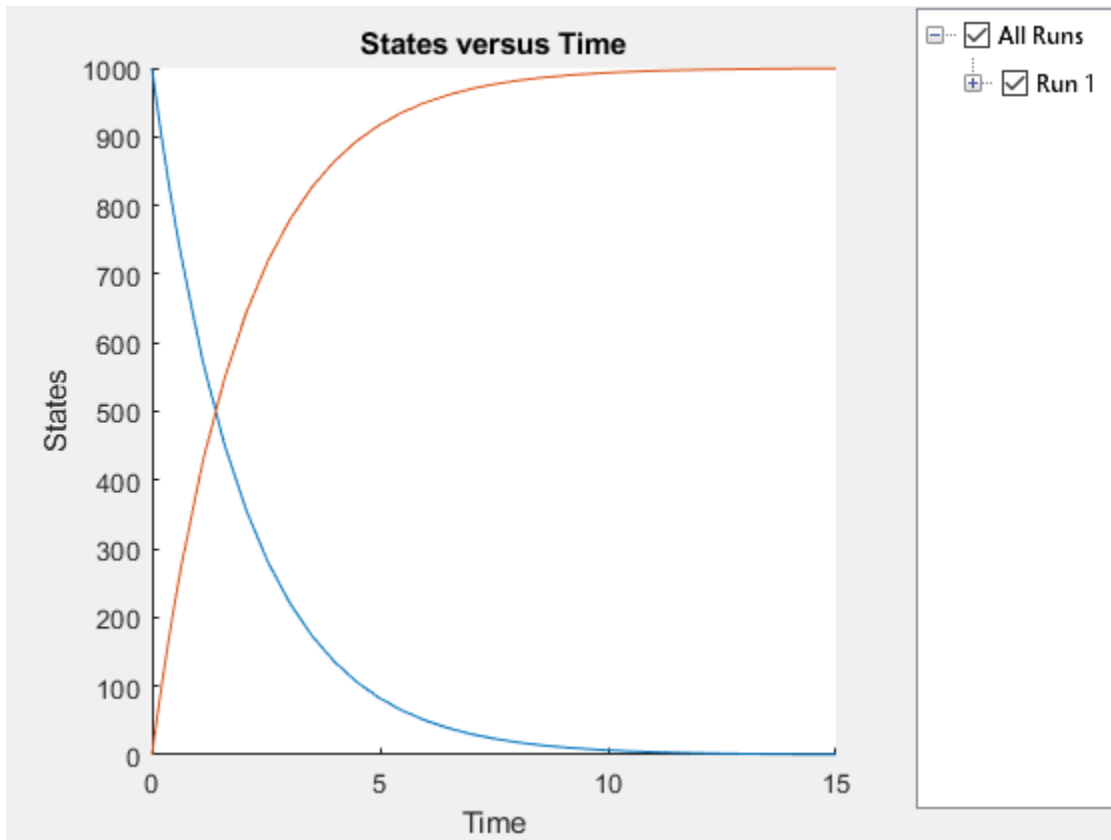
Set the configset object to be active so that its settings are used during simulation.



```
setactiveconfigset(m1,configsetObj);
```

Simulate the model and plot results.

```
simdata = sbiosimulate(m1);  
sbioplot(simdata);
```



## See Also

model object, configset object, getconfigset, removeconfigset, setactiveconfigset

**Introduced in R2006a**

## addcontent (variant)

Append content to variant object

### Syntax

```
addcontent(variantObj, contents)
```

```
addcontent(variantObj1, variantObj2)
```

### Arguments

<i>variantObj</i>	Specify the variant object to which you want to append data. The Content property is modified to add the new data.
<i>contents</i>	Specify the data you want to add to a variant object. Contents can either be a cell array or an array of cell arrays. A valid cell array should have the form { 'Type', 'Name', 'PropertyName', PropertyValue}, where <i>PropertyValue</i> is the new value to be applied for the <i>PropertyName</i> . Valid <i>Type</i> , <i>Name</i> , and <i>PropertyName</i> values are as follows.

'Type'	'Name'	'PropertyName'
'species'	Name of the species. If there are multiple species in the model with the same name, specify the species as [compartmentName.speciesName], where <i>compartmentName</i> is the name of the compartment containing the species.	'InitialAmount'
'parameter'	If the parameter scope is a model, specify the parameter name. If the parameter scope is a kinetic law, specify [reactionName.parameterName].	'Value'
'compartment'	Name of the compartment.	'Capacity'

### Description

`addcontent(variantObj, contents)` adds the data stored in the variable *contents* to the variant object (*variantObj*).

`addcontent(variantObj1, variantObj2)` appends the data in the Content property of the variant object *variantObj2* to the Content property of variant object *variantObj1*.

---

**Note** Remember to use the `addcontent` method instead of using the `set` method on the Content property because the `set` method replaces the data in the Content property, whereas `addcontent` appends the data.

---

### Examples

- 1 Create a model containing one species.

```
modelObj = sbiomodel('mymodel');  
compObj = addcompartment(modelObj, 'comp1');  
speciesObj = addspecies(compObj, 'A');
```

- 2 Add a variant object that varies the `InitialAmount` property of a species named A.

```
variantObj = addvariant(modelObj, 'v1');  
addcontent(variantObj, {'species', 'A', 'InitialAmount', 5});
```

### See Also

`addvariant`, `rmcontent`, `sbiovariant`

**Introduced in R2007b**

## adddose (model)

Add dose object to model

### Syntax

```
doseObj2 = adddose(modelObj, 'DoseName')
doseObj2 = adddose(modelObj, 'DoseName', 'DoseType')
doseObj2 = adddose(modelObj, doseObj)
```

### Arguments

<i>modelObj</i>	Model object to which you add a dose object.
<i>DoseName</i>	Name of a dose object to construct and add to a model object. <i>DoseName</i> is the value of the dose object property Name.
<i>DoseType</i>	Type of dose object to construct and add to a model object. Enter either 'schedule' or 'repeat'.
<i>doseObj</i>	Dose object to add to a model object. Created with the constructor sbiodose.

### Outputs

<i>doseObj2</i>	ScheduleDose object or RepeatDose object. A RepeatDose or ScheduleDose object defines an increase (dose) to a species amount during a simulation.
-----------------	---

## Description

Before using a dose object in a simulation, use the `adddose` method to add the dose object to a SimBiology `model` object. Then, set the `Active` dose object property to `true`.

`doseObj2 = adddose(modelObj, 'DoseName')` constructs a SimBiology `RepeatDose` object (`doseObj2`), assigns `DoseName` to the property `Name`, adds the dose object to a SimBiology model object (`modelObj`), and assigns `modelObj` to the property `Parent`.

`doseObj2 = adddose(modelObj, 'DoseName', 'DoseType')` constructs either a SimBiology `ScheduleDose` object or `RepeatDose` object (`doseObj2`).

`doseObj2 = adddose(modelObj, doseObj)` adds a SimBiology dose object (`doseObj`) to a SimBiology model object (`modelObj`), copies the dose object to a second dose object (`doseObj2`), and assigns `modelObj` to the property `Parent`. The `Active` property of `doseObj2` is set to `false` by default.

---

**Note** Alternatively, you can create a dose object using `sbiodose` as a standalone dose object, which you can apply to different models. For details, see “Creating Doses Programmatically”.

---

## Examples

### Add an Infusion Dose

This example shows how to add a constant-rate infusion dose to a one-compartment model.

#### Background

Suppose you have a one-compartment model with a species named `drug` that represents the total amount of drug in the body. The drug is removed from the body via the first-order elimination represented by the reaction `drug -> null`, with the elimination rate constant `ke`. In other words, the drug concentration versus the time profile follows the

monoexponential decline  $C_t = C_0 e^{-k_e t}$ , where  $C_t$  is the drug concentration at time  $t$ ,  $C_0$  is the initial concentration, and  $k_e$  is the elimination rate constant. This example shows how

to set up such a one-compartment model and add an infusion dose at a constant rate of 10 mg/hour for the total dose amount of 250 mg.

### Create a One-compartment Model

Create a SimBiology model named `onecomp`.

```
m1 = sbiomodel('onecomp');
```

Define the elimination of the drug from the system by adding a reaction `drug -> null` to the model.

```
r1 = addreaction(m1,'drug -> null');
```

The species `drug` is automatically created and added to the compartment. The `null` species is a reserved species that acts as a sink in this reaction.

Add a mass action kinetic law to the reaction. This kinetic law defines the drug elimination to follow the first-order kinetics.

```
k1 = addkineticlaw(r1,'MassAction');
```

Define the elimination rate parameter `ke` and add it to the kinetic law.

```
p1 = addparameter(k1,'ke','Value',1.0,'ValueUnits','1/hour');
```

Specify the rate parameter `ke` as the forward rate parameter of the reaction by setting the `ParameterVariableNames` property of kinetic law object `k1`. This allows SimBiology to determine the reaction rate for `drug -> null` reaction.

```
k1.ParameterVariableNames = 'ke';
```

### Set up an Infusion Dose

Add a dose object to the model using the `adddose` method. Specify the amount of the dose (`Amount`), the dose target (`TargetName`), and the infusion rate (`Rate`). You also need to set the `Active` property of the dose object to `true` so that the dose is applied to the model during simulation.

```
d1 = adddose(m1,'InfusionDose');  
d1.Amount = 250;  
d1.TargetName = 'drug';  
d1.Rate = 10;  
d1.RateUnits = 'milligram/hour';  
d1.Active = true;
```

### **Simulate the Model**

Change the simulation stop time to 48 hours to see the complete time course.

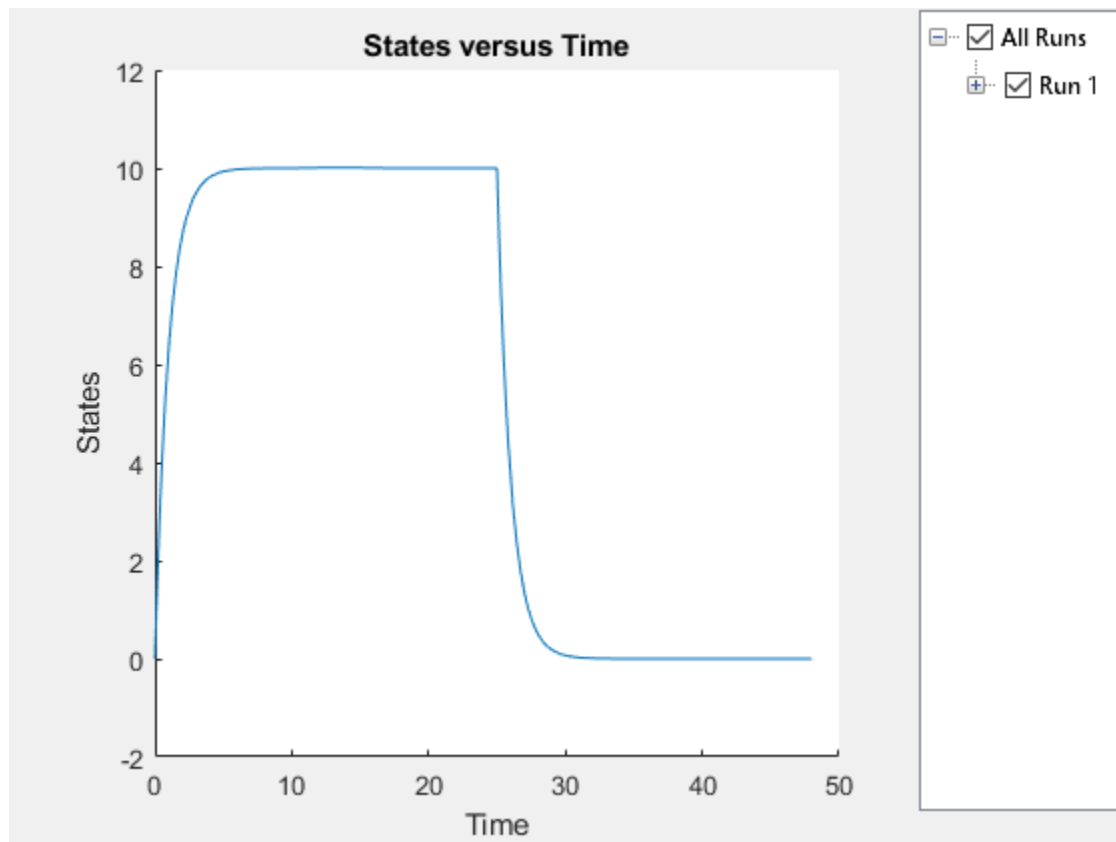
```
cs = getconfigset(m1);  
cs.StopTime = 48;  
cs.TimeUnits = 'hour';  
sd = sbiosimulate(m1);
```

### **Plot results**

Plot the concentration versus the time profile of the drug in the system.

```
sbioplot(sd);
```





## See Also

RepeatDose object | ScheduleDose object | getdose | model object | removedose | sbiodose

## Topics

"Doses"

**Introduced in R2010a**

## addevent (model)

Add event object to model object

### Syntax

```
eventObj = addevent(modelObj, 'TriggerValue', 'EventFcnsValue')
```

```
eventObj = addevent(...'PropertyName', PropertyValue...)
```

### Arguments

<i>modelObj</i>	Model object.
<i>TriggerValue</i>	Required property to specify a trigger condition. Must be a MATLAB expression that evaluates to a logical value. Use the keyword 'time' to specify that an event occurs at a specific time during the simulation. For more information, see Trigger.
<i>EventFcnsValue</i>	Character vector or a cell array of character vectors, each of which specifies an assignment of the form ' <i>objectname</i> = <i>expression</i> ', where <i>objectname</i> is the name of a valid object. Defines what occurs when the event is triggered. For more information, see EventFcns.
<i>PropertyName</i>	Property name for an event object from “Property Summary” on page 2-31.
<i>PropertyValue</i>	Property value. For more information on property values, see the property reference for each property listed in “Property Summary” on page 2-31.

### Description

`eventObj = addevent(modelObj, 'TriggerValue', 'EventFcnsValue')` creates an Event object (`eventObj`) and adds the event to the model (`modelObj`). In

the event object, this method assigns a value (*TriggerValue*) to the property *TriggerCondition*, assigns a value (*EventFcnsValue*) to the property *EventFcns*, and assigns the model object (*modelObj*) to the property *Parent*. In the model object, this method appends the event object to the property *Events*.

When the trigger expression in the property *Trigger* changes from false to true, the assignments in *EventFcns* are executed during simulation.

For details on how events are handled during a simulation, see “Events”.

`eventObj = addevent(...'PropertyName', PropertyValue...)` defines optional properties. The property name and property value pairs can be any format supported by the function `set`.

## Property Summary

Active	Indicate object in use during simulation
EventFcns	Event expression
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Trigger	Event trigger
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

### Add an Event

This example shows how to add an event to a SimBiology model.

Create a simple model with a mass action reaction  $A \rightarrow B$ , where A and B are species. Also add the reaction rate parameter, *p1*, with the parameter value of 0.5.

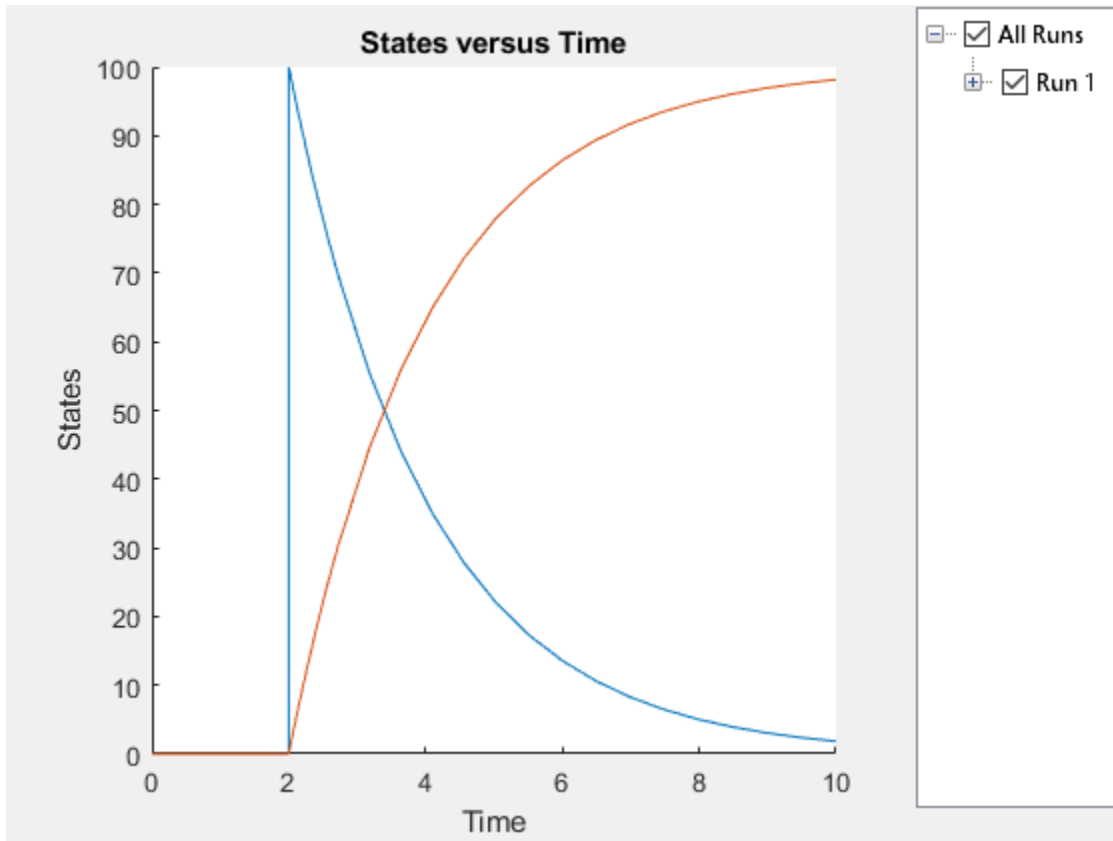
```
model      = sbiomodel('example');  
r1         = addreaction(model, 'A -> B');  
kl         = addkineticlaw(r1, 'MassAction');  
p1         = addparameter(model, 'p1', 0.5);  
kl.ParameterVariableNames = 'p1';
```

Increase the amount of species A to 100 at time = 2. You can do this by adding an event object to the model. You must specify the event trigger (`time >= 2`), and also the event function, which defines what happens when the event is triggered. In this example, the event function is `A = 100`.

```
e1 = addevent(model, 'time>=2', 'A = 100');
```

Simulate the model, and plot the result.

```
sd = sbiosimulate(model);  
sbioplot(sd);
```



## See Also

### Topics

“Deterministic Simulation of a Model Containing a Discontinuity”  
“Events”

**Introduced in R2007b**

## addkineticlaw (reaction)

Create kinetic law object and add to reaction object

### Syntax

```
kineticlawObj = addkineticlaw(reactionObj, 'KineticLawNameValue')

```

```
kineticlawObj = addkineticlaw(..., 'PropertyName',
PropertyValue, ...)
```

### Arguments

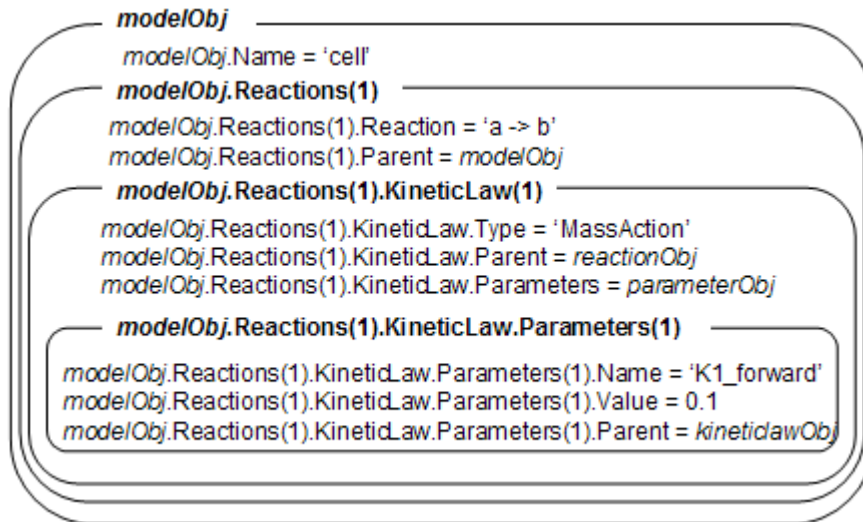
<i>reactionObj</i>	Reaction object. Enter a variable name for a reaction object.
<i>KineticLawNameValue</i>	<p>Property to select the type of kinetic law object to create. For built-in kinetic law, valid values are:</p> <p>'Unknown', 'MassAction', 'Henri-Michaelis-Menten', 'Henri-Michaelis-Menten-Reversible', 'Hill-Kinetics', 'Iso-Uni-Uni', 'Ordered-Bi-Bi', 'Ping-Pong-Bi-Bi', 'Competitive-Inhibition', 'NonCompetitive-Inhibition', and 'UnCompetitive-Inhibition'.</p> <p>Find valid <i>KineticLawNameValue</i> by using sbioroot to create a SimBiology root object, then query the object with the commands  <code>rootObj.BuiltinLibrary.KineticLaws</code> and  <code>rootObj.UserDefinedLibrary.KineticLaws</code>.</p> <p><code>sbiowhos -kineticlaw</code> lists kinetic laws in the SimBiology root, which includes kinetic laws from both the <code>BuiltInLibrary</code> and the <code>UserDefinedLibrary</code>.</p>

## Description

*kineticlawObj* = `addkineticlaw(reactionObj, 'KineticLawNameValue')` creates and adds a `KineticLaw` object to the *reactionObj*.

In the kinetic law object, this method assigns a name (*KineticLawNameValue*) to the property `KineticLawName` and assigns the reaction object to the property `Parent`. In the reaction object, this method assigns the kinetic law object to the property `KineticLaw`.

```
modelObj = sbiomodel('cell');
reactionObj = addreaction(modelObj, 'a -> b');
kineticlawObj = addkineticlaw(reactionObj, 'MassAction');
parameterObj = addparameter(kineticlawObj, 'K1_forward', 0.1);
set(kineticlawObj, ParameterVariableName, 'K1_forward');
```



*KineticLawNameValue* is any valid kinetic law definition. See “Kinetic Law Definition” on page 3-88 for a definition of kinetic laws and more information about how they are used to get the reaction rate expression.

*kineticlawObj* = `addkineticlaw(..., 'PropertyName', PropertyValue, ...)` constructs a kinetic law object, *kineticlawObj*, and

configures *kineticLawObj* with property value pairs. The property name/property value pairs can be in any format supported by the function `set`. The `kineticLawObj` properties are listed in “Property Summary” on page 2-36.

---

**Note** To define a Hill kinetic rate equation with a non-integer exponent that is compatible with `DimensionalAnalysis`, see “Define a Custom Hill Kinetic Law that Works with Dimensional Analysis” on page 3-179.

---

## Property Summary

Properties for kinetic law objects

Expression (AbstractKineticLaw, KineticLaw)	Expression to determine reaction rate equation
KineticLawName	Name of kinetic law applied to reaction
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parameters	Array of parameter objects
ParameterVariableNames	Cell array of reaction rate parameters
ParameterVariables	Parameters in kinetic law definition
Parent	Indicate parent object
SpeciesVariableNames	Cell array of species in reaction rate equation
SpeciesVariables	Species in abstract kinetic law
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples



## Convert Substrate into Product Using Henri-Michaelis-Menten Kinetics

This example shows how to simulate the conversion of a substrate into a product using the Henri-Michaelis-Menten enzyme kinetics.

Create a model named `mymodel`.

```
model = sbiomodel('mymodel');
```

Add a reaction that represents the conversion of a substrate to a product.

```
reaction = addreaction(model, 'Substrate -> Product');
```

Add the built-in Henri-Michaelis-Menten kinetic law to the reaction.

```
kineticLaw = addkineticlaw(reaction, 'Henri-Michaelis-Menten');
kineticLaw.Expression
```

```
ans =
'Vm*S/(Km + S)'
```

The kinetic law has two parameters and a species that you need to define. View these parameters.

```
kineticLaw.ParameterVariables
```

```
ans = 1x2 cell array
    {'Vm'}    {'Km'}
```

```
kineticLaw.SpeciesVariables
```

```
ans = 1x1 cell array
    {'S'}
```

To define the parameters, create two parameter objects and set parameter values.

```
Vm_param = addparameter(kineticLaw, 'Vm_param', 'Value', 6.0);
Km_param = addparameter(kineticLaw, 'Km_param', 'Value', 1.25);
```

Map the parameters accordingly by setting the `ParameterVariableNames` property. This associates the parameters in the expression with the two parameters you just created using a one-to-one mapping in the order given.

```
kineticLaw.ParameterVariableNames = {'Vm_param', 'Km_param'};
```

Also associate the Substrate species with the species S in the expression.

```
kineticLaw.SpeciesVariableNames = {'Substrate'};
```

Verify the mapping by looking at the reaction rate and checking the parameters and species are correctly substituted according to the expression.

```
reaction.ReactionRate
```

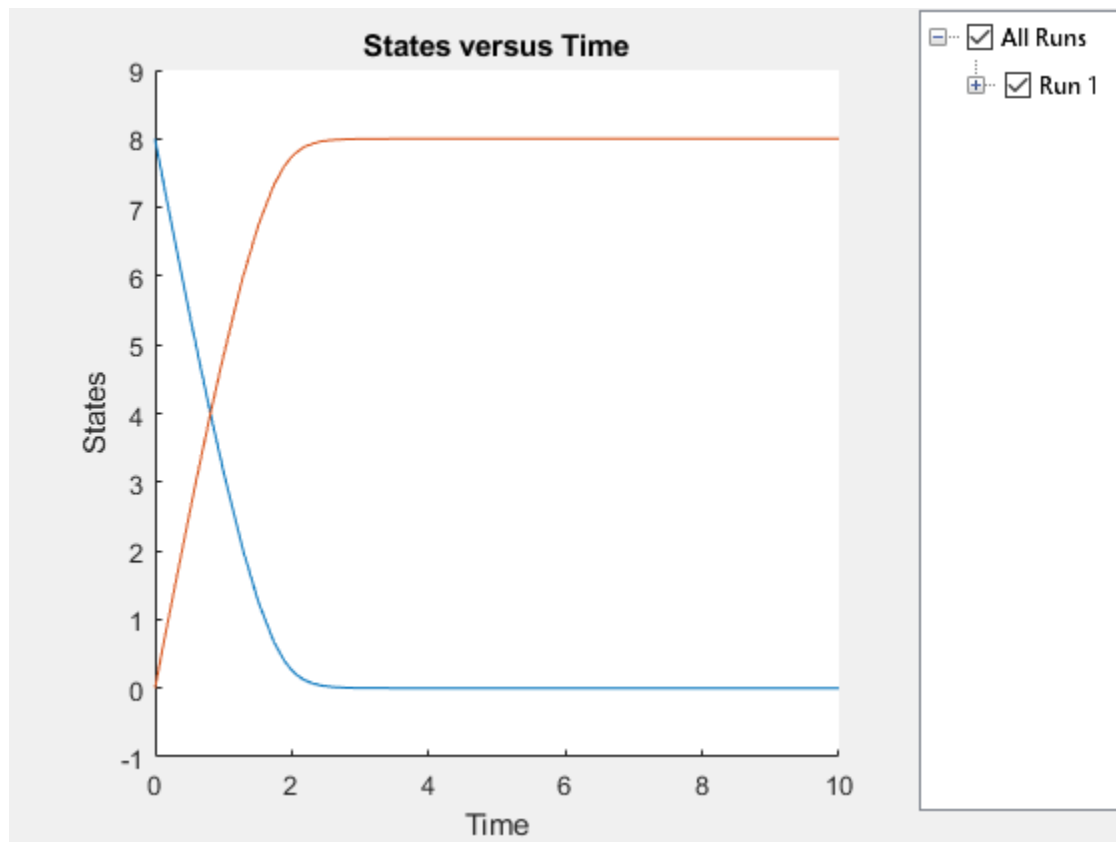
```
ans =  
'Vm_param*Substrate/(Km_param+Substrate)'
```

Enter the initial amount of the substrate species for simulation.

```
model.Species(1).InitialAmount = 8;
```

Simulate the model and plot results.

```
simdata = sbiosimulate(model);  
sbioplot(simdata);
```



## See Also

addreaction, setparameter

**Introduced in R2006a**

## addparameter (model, kineticlaw)

Create parameter object and add to model or kinetic law object

### Syntax

```
parameterObj = addparameter(Obj, 'NameValue')  
parameterObj = addparameter(Obj, 'NameValue', ValueValue)  
  
parameterObj = addparameter(...'PropertyName', PropertyValue...)
```

### Arguments

<i>Obj</i>	Model object or kineticlaw object. Enter a variable name for the object.
<i>NameValue</i>	Property for a parameter object. Enter a unique character vector.  Since objects can use this property to reference a parameter, a parameter object must have a unique name at the level it is created. For example, a kinetic law object cannot contain two parameter objects named kappa. However, the model object that contains the kinetic law object can contain a parameter object named kappa along with the kinetic law object.  For information on naming parameters, see Name.
<i>ValueValue</i>	Property for a parameter object. Enter a number.

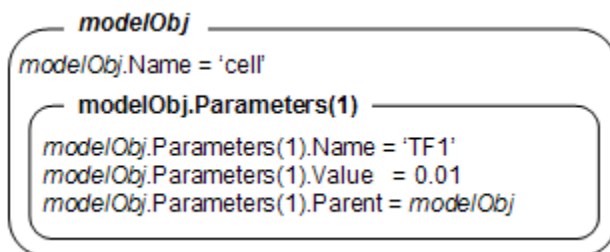
### Description

`parameterObj = addparameter(Obj, 'NameValue')` creates a parameter object and returns the object (*parameterObj*). In the parameter object, this method assigns a value (*NameValue*) to the property *Name*, assigns a value 1 to the property *Value*, and

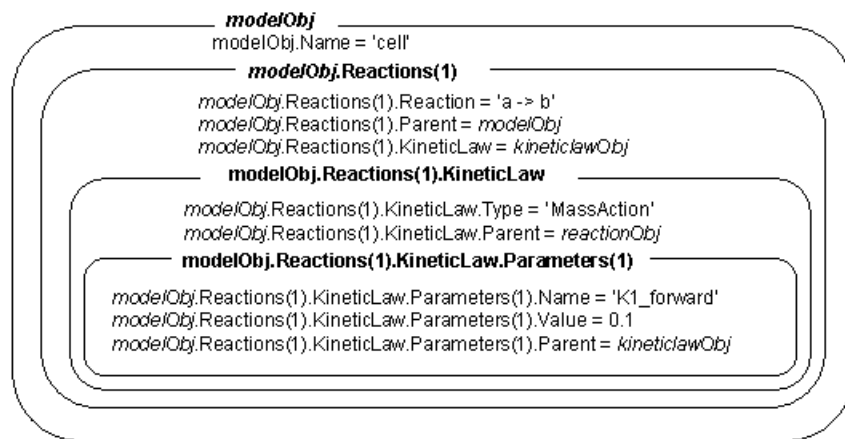
assigns the model or kinetic law object to the property `Parent`. In the model or kinetic law object, (*Obj*), this method assigns the parameter object to the property `Parameters`.

A parameter object defines an assignment that a model or a kinetic law can use. The scope of the parameter is defined by the parameter parent. If a parameter is defined with a kinetic law object, then only the kinetic law object and objects within the kinetic law object can use the parameter. If a parameter object is defined with a model object as its parent, then all objects within the model (including all rules, events and kinetic laws) can use the parameter.

```
modelObj = sbiomodel('cell')
parameterObj = addparameter(modelObj, 'TF1', 0.01)
```



```
modelObj = sbiomodel('cell')
reactionObj = addreaction(modelObj, 'a -> b')
kineticlawObj = addkineticlaw (reactionObj, 'MassAction')
parameterObj = addparameter(kineticlawObj, 'K1_forward', 0.1)
```



`parameterObj = addparameter(Obj, 'NameValue', ValueValue)` creates a parameter object, assigns a value (*NameValue*) to the property *Name*, assigns the value (*ValueValue*) to the property *Value*, and assigns the `model` object or the `kineticlaw` object to the property *Parent*. In the `model` or `kinetic law` object (`Obj`), this method assigns the parameter object to the property *Parameters*, and returns the parameter object to a variable (`parameterObj`).

`parameterObj = addparameter(...'PropertyName', PropertyValue...)` defines optional property values. The name-value pairs can be in any format supported by the function `set`.

**Scope of a parameter** — A parameter can be *scoped* to either a model or a kinetic law.

- When a kinetic law searches for a parameter in its expression, it first looks in the parameter list of the kinetic law. If the parameter isn't found there, it moves to the model that the kinetic law object is in and looks in the model parameter list. If the parameter isn't found there, it moves to the model parent.
- When a rule searches for a parameter in its expression, it looks in the parameter list for the model. If the parameter isn't found there, it moves to the model parent. A rule cannot use a parameter that is scoped to a kinetic law. So for a parameter to be used in both a reaction rate equation and a rule, the parameter should be *scoped* to a model.

Additional parameter object properties can be viewed with the `get` command. Additional parameter object properties can be modified with the `set` command. The parameters of `Obj` can be viewed with `get(Obj, 'Parameters')`.

A SimBiology parameter object can be copied to a SimBiology model or kinetic law object with `copyobj`. A SimBiology parameter object can be removed from a SimBiology model or kinetic law object with `delete`.

## Method Summary

Methods for parameter objects

copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
findUsages (species, parameter, compartment)	Find out how a species, parameter, or compartment is used in a model
get (any object)	Get object properties
rename (compartment, parameter, species, reaction)	Rename object and update expressions
set (any object)	Set object properties

## Property Summary

Properties for parameter objects

ConstantValue	Specify variable or constant parameter value
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object
Value	Assign value to parameter object
ValueUnits	Parameter value units

## Example

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Define a kinetic law for the reaction object.

```
kineticlawObj = addkineticlaw(reactionObj, 'MassAction');
```

- 3 Add a parameter and assign it to the kinetic law object (kineticlawObj); add another parameter and assign to the model object (modelObj).

```
% Add parameter to kinetic law object
parameterObj1 = addparameter (kineticlawObj, 'K1');

get (kineticlawObj, 'Parameters')
```

MATLAB returns:

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	K1	1	

```
% Add parameter with value 0.9 to model object
parameterObj1 = addparameter (modelObj, 'K2', 0.9);
```

```
get (modelObj, 'Parameters')
```

MATLAB returns:

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	K2	1	

## See Also

model object, kineticlaw object, addreaction

**Introduced in R2006a**



## addproduct (reaction)

Add product species object to reaction object

### Syntax

```
speciesObj = addproduct(reactionObj, 'NameValue')
speciesObj = addproduct(reactionObj, speciesObj)
speciesObj = addproduct(reactionObj, 'NameValue', Stoichcoefficient)
speciesObj = addproduct(reactionObj, speciesObj, Stoichcoefficient)
```

### Arguments

<i>reactionObj</i>	Reaction object. Enter a name for the reaction object.
<i>NameValue</i>	Property of a species object that names the object (not the reaction object). Enter a unique character vector. For example, 'fructose 6-phosphate'.  A species object can be referenced by other objects using this property. You can use the function <code>sbiiselect</code> on page 1-274 to find an object with a specific <code>NameValue</code> .
<i>speciesObj</i>	Species object.
<i>Stoichcoefficient</i>	Stoichiometric coefficients for products, length of array equal to length of <code>NameValue</code> , or length of <i>speciesObj</i> .

### Description

`speciesObj = addproduct(reactionObj, 'NameValue')` creates a species object and returns the species object (*speciesObj*). In the species object, this method assigns the value (*NameValue*) to the property `Name`. In the reaction object, this method assigns the species object to the property `Products`, modifies the reaction equation in the property `Reaction` to include the new species, and adds the stoichiometric coefficient 1 to the property `Stoichiometry`.

When you define a reaction with a new species:

- If no compartment objects exist in the model, the method creates a compartment object (called '*unnamed*') in the model and adds the newly created species to that compartment.
- If only one compartment object (*compObj*) exists in the model, the method creates a species object in that compartment.
- If there is more than one compartment object (*compObj*) in the model, you must qualify the species name with the compartment name.

For example, `cell.glucose` denotes that you want to put the species named `glucose` into a compartment named `cell`. Additionally, if the compartment named `cell` does not exist, the process of adding the reaction creates the compartment and names it `cell`.

Create and add a species object to a compartment object with the method `addspecies` on page 2-61.

`speciesObj = addproduct(reactionObj, speciesObj)`, in the species object (`speciesObj`), assigns the parent object of the `reactionObj` to the species property `Parent`. In the reaction object (`reactionObj`), it assigns the species object to the property `Products`, modifies the reaction equation in the property `Reaction` to include the new species, and adds the stoichiometric coefficient 1 to the property `Stoichiometry`.

`speciesObj = addproduct(reactionObj, 'NameValue', Stoichcoefficient)`, in addition to the description above, adds the stoichiometric coefficient (`Stoichcoefficient`) to the property `Stoichiometry`. If `NameValue` is a cell array of species names, then `Stoichcoefficient` must be a vector of doubles with the same length as `NameValue`.

`speciesObj = addproduct(reactionObj, speciesObj, Stoichcoefficient)`, in addition to the description above, adds the stoichiometric coefficient (`Stoichcoefficient`) to the property `Stoichiometry`.

Species names are referenced by reaction objects, kinetic law objects, and model objects. If you change the `Name` of a species the reaction also uses the new name. You must however configure all other applicable elements such as rules that use the species, and the kinetic law object.

## Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'A + C -> U');
```

- 2 Modify the reaction of the reactionObj from  $A + C \rightarrow U$  to  $A + C \rightarrow U + 2 H$ .

```
speciesObj = addproduct(reactionObj, 'H', 2);
```

## See Also

addspecies

**Introduced in R2006a**

## addreactant (reaction)

Add species object as reactant to reaction object

### Syntax

```
speciesObj = addreactant(reactionObj, 'NameValue')
addreactant(reactionObj, speciesObj, StoichCoefficient)
addreactant(reactionObj, 'NameValue', StoichCoefficient)
```

### Arguments

<i>reactionObj</i>	Reaction object.
<i>NameValue</i>	Name property of a species object. Enter a unique character vector, for example, 'glucose'. A species object can be referenced by other objects using this property.  You can use the function <code>sbiiselect</code> on page 1-274 to find an object with a specific Name property value.
<i>speciesObj</i>	Species object or cell array of species objects.
<i>StoichCoefficient</i>	Stoichiometric coefficients for reactants, length of array equal to length of <i>NameValue</i> or length of <i>speciesObj</i> .

### Description

`speciesObj = addreactant(reactionObj, 'NameValue')` creates a species object and returns the species object (*speciesObj*). In the species object, this method assigns the value (*NameValue*) to the property *Name*. In the reaction object, this method assigns the species object to the property *Reactants*, modifies the reaction equation in the property *Reaction* to include the new species, and adds the stoichiometric coefficient -1 to the property *Stoichiometry*.

When you define a reaction with a new species:

- If no compartment objects exist in the model, the method creates a compartment object (called '*unnamed*') in the model and adds the newly created species to that compartment.
- If only one compartment object (*compObj*) exists in the model, the method creates a species object in that compartment.
- If there is more than one compartment object (*compObj*) in the model, you must qualify the species name with the compartment name.

For example, `cell.glucose` denotes that you want to put the species named `glucose` into a compartment named `cell`. Additionally, if the compartment named `cell` does not exist, the process of adding the reaction creates the compartment and names it `cell`.

Create and add a species object to a compartment object with the method `addspecies` on page 2-61.

`addreactant(reactionObj, speciesObj, StoichCoefficient)`, in the species object (*speciesObj*), assigns the parent object to the `speciesObj` property `Parent`. In the reaction object (*reactionObj*), it assigns the species object to the property `Reactants`, modifies the reaction equation in the property `Reaction` to include the new species, and adds the stoichiometric coefficient `-1` to the property `Stoichiometry`. If *speciesObj* is a cell array of species objects, then *StoichCoefficient* must be a vector of doubles with the same length as *speciesObj*.

`addreactant(reactionObj, 'NameValue', StoichCoefficient)`, in addition to the description above, adds the stoichiometric coefficient (*StoichCoefficient*) to the property `Stoichiometry`. If *NameValue* is a cell array of species names, then *StoichCoefficient* must be a vector of doubles with the same length as *NameValue*.

Species names are referenced by reaction objects, kinetic law objects, and model objects. If you change the `Name` of a species the reaction also uses the new name. You must, however, configure all other applicable elements such as rules that use the species, and the kinetic law object.

See "Specifying Species Names in SimBiology" for more information on species names.

### Example

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'A -> U');
```

- 2 Modify the reaction of the `reactionObj` from  $A \rightarrow U$  to be  $A + 3 C \rightarrow U$ .

```
speciesObj = addreactant(reactionObj, 'C', 3);
```

### See Also

`addspecies`

**Introduced in R2006a**

## addreaction (model)

Create reaction object and add to model object

### Syntax

```

reactionObj = addreaction(modelObj, 'ReactionValue')
reactionObj = addreaction(modelObj, 'ReactantsValue',
'ProductsValue')
reactionObj = addreaction(modelObj, 'ReactantsValue',
RStoichCoefficients, 'ProductsValue', PStoichCoefficients)

reactionObj = addreaction(...'PropertyName', PropertyValue...)

```

### Arguments

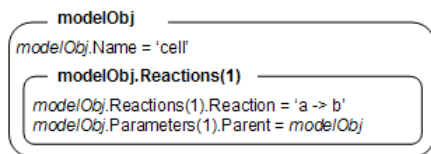
<i>modelObj</i>	SimBiology model object.
<i>ReactionValue</i>	<p>Specify the reaction equation. Enter a character vector. A hyphen preceded by a space and followed by a right angle bracket (-&gt;) indicates reactants going forward to products. A hyphen with left and right angle brackets (&lt;-&gt;) indicates a reversible reaction. Coefficients before reactant or product names must be followed by a space.</p> <p>Examples are 'A -&gt; B', 'A + B -&gt; C', '2 A + B -&gt; 2 C', and 'A &lt;-&gt; B'. Enter reactions with spaces between the species.</p> <p>If there are multiple compartments, or to specify the compartment name, use <i>compartmentName.speciesName</i>.</p> <p>Examples are 'cytoplasm.A -&gt; cytoplasm.B', 'cytoplasm.A -&gt; nucleus.A', and 'cytoplasm.A + cytoplasm.B -&gt; nucleus.AB'.</p>

<i>ReactantsValue</i>	Character vector defining the species name, a cell array of character vectors, a species object, or an array of species objects. If using names, qualify with compartment names if there are multiple compartments.
<i>ProductsValue</i>	Character vector defining the species name, a cell array of character vectors, a species object, or an array of species objects. If using names, qualify with compartment names if there are multiple compartments.
<i>RStoichCoefficients</i>	Stoichiometric coefficients for reactants, length of array equal to length of <i>ReactantsValue</i> .
<i>PStoichCoefficients</i>	Stoichiometric coefficients for products, length of array equal to length of <i>ProductsValue</i> .

## Description

`reactionObj = addreaction(modelObj, 'ReactionValue')` creates a reaction object, assigns a value (*ReactionValue*) to the property `Reaction`, assigns reactant species object(s) to the property `Reactants`, assigns the product species object(s) to the property `Products`, and assigns the `model` object to the property `Parent`. In the `Model` object (`modelObj`), this method assigns the reaction object to the property `Reactions`, and returns the reaction object (`reactionObj`).

```
reactionObj = addreaction(modelObj, 'a -> b')
```



When you define a reaction with a new species:

- If no compartment objects exist in the model, the method creates a compartment object (called '*unnamed*') in the model and adds the newly created species to that compartment.
- If only one compartment object (`compObj`) exists in the model, the method creates a species object in that compartment.



- If there is more than one compartment object (`compObj`) in the model, you must qualify the species name with the compartment name.

For example, `cell.glucose` denotes that you want to put the species named `glucose` into a compartment named `cell`. Additionally, if the compartment named `cell` does not exist, the process of adding the reaction creates the compartment and names it `cell`.

You can manually add a species to a compartment object with the method `addspecies`.

You can add species to a reaction object using the methods `addreactant` or `addproduct`. You can remove species from a reaction object with the methods `rmreactant` or `rmproduct`. The property `Reaction` is modified by adding or removing species from the reaction equation.

You can copy a SimBiology reaction object to a model object with the function `copyobj`. You can remove the SimBiology reaction object from a SimBiology model object with the function `delete`.

You can view additional reaction object properties with the `get` command. For example, the reaction equation of `reactionObj` can be viewed with the command `get(reactionObj, 'Reaction')`. You can modify additional reaction object properties with the command `set`.

`reactionObj = addreaction(modelObj, 'ReactantsValue', 'ProductsValue')` creates a reaction object, assigns a value to the property `Reaction` using the reactant (`ReactantsValue`) and product (`ProductsValue`) names, assigns the species objects to the properties `Reactants` and `Products`, and assigns the model object to the property `Parent`. In the model object (`modelObj`), this method assigns the reaction object to the property `Reactions`, and returns the reaction object (`reactionObj`). The stoichiometric values are assumed to be 1.

`reactionObj = addreaction(modelObj, 'ReactantsValue', RStoichCoefficients, 'ProductsValue', PStoichCoefficients)` adds stoichiometric coefficients (`RStoichCoefficients`) for reactant species, and stoichiometric coefficients (`PStoichCoefficients`) for product species to the property `Stoichiometry` on page 3-224. The length of `Reactants` and `RCoefficients` must be equal, and the length of `Products` and `PCoefficients` must be equal.

`reactionObj = addreaction(...'PropertyName', PropertyValue...)` defines

optional properties. The property name/property value pairs can be in any format supported by the function set.

---

**Note** If you use the `addreaction` method to create a reaction rate expression that is not continuous and differentiable, see “Using Events to Address Discontinuities in Rule and Reaction Rate Expressions” before simulating your model.

---

## Method Summary

Methods for reaction objects

<code>addkineticlaw</code> (reaction)	Create kinetic law object and add to reaction object
<code>addproduct</code> (reaction)	Add product species object to reaction object
<code>addreactant</code> (reaction)	Add species object as reactant to reaction object
<code>copyobj</code> (any object)	Copy SimBiology object and its children
<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>get</code> (any object)	Get object properties
<code>rename</code> (compartment, parameter, species, reaction)	Rename object and update expressions
<code>rmproduct</code> (reaction)	Remove species object from reaction object products
<code>rmreactant</code> (reaction)	Remove species object from reaction object reactants
<code>set</code> (any object)	Set object properties

## Property Summary

Properties for reaction objects

Active	Indicate object in use during simulation
KineticLaw	Show kinetic law used for ReactionRate
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Products	Array of reaction products
Reactants	Array of reaction reactants
Reaction	Reaction object reaction
ReactionRate	Reaction rate equation in reaction object
Reversible	Specify whether reaction is reversible or irreversible
Stoichiometry	Species coefficients in reaction
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

Create a model, add a reaction object, and assign the expression for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object, of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

reactionObj KineticLaw property is configured to kineticlawObj.

- 3 The 'Henri-Michaelis-Menten' kinetic law has two parameter variables (V<sub>m</sub> and K<sub>m</sub>) and one species variable (S) that should to be set. To set these variables, first create the parameter variables as parameter objects (parameterObj1, parameterObj2) with names V<sub>m\_d</sub>, and K<sub>m\_d</sub>, and assign the objects Parent property value to the kineticlawObj.

```
parameterObj1 = addparameter(kineticlawObj, 'Vm_d');  
parameterObj2 = addparameter(kineticlawObj, 'Km_d');
```

- 4 Set the variable names for the kinetic law object.

```
set(kineticlawObj, 'ParameterVariableNames', {'Vm_d' 'Km_d'});  
set(kineticlawObj, 'SpeciesVariableNames', {'a'});
```

- 5 Verify that the reaction rate is expressed correctly in the reaction object ReactionRate property.

```
get (reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =
```

```
Vm_d*a/(Km_d+a)
```

## See Also

model object, addkineticlaw, addproduct, addreactant, rmproduct, rmreactant

**Introduced in R2006a**

## addrule (model)

Create rule object and add to model object

### Syntax

```
ruleObj = addrule(modelObj, Rule)
```

```
ruleObj = addrule(modelObj, Rule, RuleType)
```

```
ruleObj = addrule(..., 'PropertyName', PropertyValue, ...)
```

### Arguments

<i>modelObj</i>	Model object to which to add the rule.
<i>Rule</i>	Character vector specifying the rule. For example, enter the algebraic rule ' $V_a * E_a + V_i * E_i - K_2$ '.
<i>RuleType</i>	Character vector specifying the type of rule. Choices are: <ul style="list-style-type: none"> <li>• 'algebraic'</li> <li>• 'initialAssignment'</li> <li>• 'repeatedAssignment'</li> <li>• 'rate'</li> </ul> For more information, see RuleType

### Description

A rule is a mathematical expression that changes the amount of a species or the value of a parameter. It also defines how species and parameters interact with one another.

*ruleObj* = addrule(*modelObj*, *Rule*) constructs and returns *ruleObj*, a rule object. In *ruleObj*, the rule object, this method assigns the *modelObj* input argument to the Parent property, assigns the *Rule* input argument to the Rule property, and assigns

'initialAssignment' or 'algebraic' to the `RuleType` property. (This method assigns 'initialAssignment' for all assignment rules and 'algebraic' for all other rules.) In `modelObj`, the model object, this method assigns `ruleObj`, the rule object, to the `Rules` property.

`ruleObj = addrule(modelObj, Rule, RuleType)` in addition to the assignments above, assigns the `RuleType` input argument to the `RuleType` property. For more information on the types of rules, see `RuleType`.

`ruleObj = addrule(..., 'PropertyName', PropertyValue, ...)` defines optional properties. The property name/property value pairs can be in any format supported by the function `set`.

View additional rule properties with the function `get`, and modify rule properties with the function `set`. Copy a rule object to a model with the function `copyobj` on page 2-95, or delete a rule object from a model with the function `delete` on page 2-137.

---

**Note** If you use the `addrule` method to create an algebraic rule, rate rule, or repeated assignment rule, and the rule expression is not continuous and differentiable, see “Using Events to Address Discontinuities in Rule and Reaction Rate Expressions” before simulating your model.

---

## Method Summary

Methods for rule objects

<code>copyobj</code> (any object)	Copy SimBiology object and its children
<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>get</code> (any object)	Get object properties
<code>set</code> (any object)	Set object properties

## Property Summary

Properties for rule objects

Active	Indicate object in use during simulation
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Rule	Specify species and parameter interactions
RuleType	Specify type of rule for rule object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

Add a rule with the default RuleType.

- 1 Create a model object, and then add a rule object.

```
modelObj = sbiomodel('cell');
ruleObj = addrule(modelObj, '0.1*B-A')
```

- 2 Get a list of properties for a rule object.

```
get(modelObj.Rules(1)) or get(ruleObj)
```

MATLAB displays a list of rule properties.

```
Active: 1
Annotation: ''
Name: ''
Notes: ''
Parent: [1x1 SimBiology.Model]
Rule: '0.1*B-A'
RuleType: 'algebraic'
Tag: ''
Type: 'rule'
UserData: []
```

Add a rule with the RuleType property set to rate.

- 1 Create model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a -> b');
```

- 2 Add a rule which defines that the quantity of a species c. In the rule expression, k is the rate constant for a -> b.

```
ruleObj = addrule(modelObj, 'c = k*(a+b)')
```

- 3 Change the RuleType from default ('algebraic') to 'rate', and verify using the get command.

```
set(ruleObj, 'RuleType', 'rate');  
get(ruleObj)
```

MATLAB returns all the properties for the rule object.

```
Active: 1  
Annotation: ''  
Name: ''  
Notes: ''  
Parent: [1x1 SimBiology.Model]  
Rule: 'c = k*(a+b)'  
RuleType: 'rate'  
Tag: ''  
Type: 'rule'  
UserData: []
```

## See Also

model object, copyobj, delete, sbiomodel

**Introduced in R2006a**



## addspecies (model, compartment)

Create species object and add to compartment object within model object

### Syntax

```
speciesObj = addspecies(compObj, 'NameValue')
speciesObj = addspecies(compObj, 'NameValue', InitialAmountValue)
speciesObj = addspecies(modelObj, 'NameValue')
speciesObj = addspecies(modelObj, 'NameValue', InitialAmountValue)

speciesObj = addspecies(...'PropertyName', PropertyValue...)
```

### Arguments

<i>compObj</i>	Compartment object.
<i>modelObj</i>	Model object containing zero or one compartment.
<i>NameValue</i>	Name for a species object. Enter a character vector unique among species within <i>modelObj</i> or <i>compObj</i> . Species objects are identified by name within the Event, ReactionRate, and Rule properties.  For information on naming species, see Name.  You can use the function <code>sbiiselect</code> on page 1-274 to find an object with a specific Name property value.
<i>InitialAmountValue</i>	Initial amount value for the species object. Enter double. Positive real number, default is 0.
<i>PropertyName</i>	Enter the name of a valid property. Valid property names are listed in "Property Summary" on page 2-64.
<i>PropertyValue</i>	Enter the value for the property specified in <i>PropertyName</i> . Valid property values are listed on each property reference page.

## Description

*speciesObj* = `addspecies(compObj, 'NameValue')` creates *speciesObj*, a species object, and adds it to *compObj*, a compartment object. In the species object, this method assigns *NameValue* to the Name property, assigns *compObj* to the Parent property, and assigns 0 to the InitialAmount property. In the compartment object, this method adds the species object to the Species property.

*speciesObj* = `addspecies(compObj, 'NameValue', InitialAmountValue)`, in addition to the above, assigns *InitialAmountValue* to the InitialAmount property for the species object.

*speciesObj* = `addspecies(modelObj, 'NameValue')` creates *speciesObj*, a species object, and adds it to *compObj*, the compartment object in *modelObj*, a model object. If *modelObj* does not contain any compartments, it creates *compObj* with a Name property of 'unnamed'. In the species object, this method assigns *NameValue* to the Name property, assigns *compObj* to the Parent property, and assigns 0 to the InitialAmount property. In the compartment object, this method adds the species object to the Species property.

*speciesObj* = `addspecies(modelObj, 'NameValue', InitialAmountValue)`, in addition to the above, assigns *InitialAmountValue* to the InitialAmount property for the species object.

You can also add a species to a reaction using the methods `addreactant` on page 2-48 and `addproduct` on page 2-45.

A species object must have a unique name at the level at which it is created. For example, a compartment object cannot contain two species objects named H2O. However, another compartment can have a species named H2O.

View properties for a species object with the `get` command, and modify properties for a species object with the `set` command. You can view a summary table of species objects in a compartment (*compObj*) with `get(compObj, 'Species')` or the properties of the first species with `get(compObj.Species(1))`.

*speciesObj* = `addspecies(...'PropertyName', PropertyValue...)` defines optional properties. The property name/property value pairs can be in any format

supported by the function `set` (for example, name-value pairs, structures, and name-value cell array pairs). The property summary on this page shows the list of properties.

If there is more than one compartment object (`compObj`) in the model, you must qualify the species name with the compartment name. For example, `cell.glucose` denotes that you want to put the species named `glucose` into a compartment named `cell`. Additionally, if the compartment named `cell` does not exist, the process of adding the reaction creates the compartment and names it `cell`.

If you change the name of a species you must configure all applicable elements, such as events and rules that use the species, any user-specified `ReactionRate`, or the kinetic law object property `SpeciesVariableNames`. Use the method `setspecies` to configure `SpeciesVariableNames`.

To update species names in the SimBiology graphical user interface, access each appropriate pane through the **Project Explorer**. You can also use the **Find** feature to locate the names that you want to update. The **Output** pane opens with the results of **Find**. Double-click a result row to go to the location of the model component.

Species names are automatically updated for reactions that use `MassAction` kinetic law.

## Method Summary

Methods for species objects

<code>copyobj</code> (any object)	Copy SimBiology object and its children
<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>findUsages</code> (species, parameter, compartment)	Find out how a species, parameter, or compartment is used in a model
<code>get</code> (any object)	Get object properties
<code>rename</code> (compartment, parameter, species, reaction)	Rename object and update expressions
<code>set</code> (any object)	Set object properties

## Property Summary

Properties for species objects

BoundaryCondition	Indicate species boundary condition
ConstantAmount	Specify variable or constant species amount
InitialAmount	Species initial amount
InitialAmountUnits	Species initial amount units
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

Add two species to a model, where one is a reactant and the other is the enzyme catalyzing the reaction.

- 1 Create a model object named `my_model` and add a compartment object.

```
modelObj = sbiomodel ('my_model');  
compObj = addcompartment(modelObj, 'comp1');
```

- 2 Add two species objects named `glucose_6_phosphate` and `glucose_6_phosphate_dehydrogenase`.

```
speciesObj1 = addspecies (compObj, 'glucose_6_phosphate');  
speciesObj2 = addspecies (compObj, ...  
                        'glucose_6_phosphate_dehydrogenase');
```

- 3 Set the initial amount of `glucose_6_phosphate` to 100 and verify.

```
set (speciesObj1, 'InitialAmount', 100);  
get (speciesObj1, 'InitialAmount')
```

MATLAB returns:

```
ans =
```

```
100
```

- 4 Use `get` to note that `modelObj` contains the species object array.

```
get(compObj, 'Species')
```

MATLAB returns:

```
SimBiology Species Array
```

```
Index: Name: InitialAmount: InitialAmountUnits:
1 glucose_6_phosphate 100
2 glucose_6_phosphate_dehydrogenase 0
```

- 5 Retrieve information about the first species in the array.

```
get(compObj.Species(1))
    Annotation: ''
    BoundaryCondition: 0
    ConstantAmount: 0
    InitialAmount: 100
    InitialAmountUnits: ''
    Name: 'glucose_6_phosphate'
    Notes: ''
    Parent: [1x1 SimBiology.Compartment]
    Tag: ''
    Type: 'species'
    UserData: []
```

## See Also

`model` object, `compartment` object, `addcompartment`, `addproduct`, `addreactant`, `addreaction`, `get`, `set`

**Introduced in R2006a**

## addvariant (model)

Add variant to model

### Syntax

```
variantObj = addvariant(modelObj, 'NameValue')  
variantObj2 = addvariant(modelObj, variantObj)
```

### Arguments

<i>modelObj</i>	Specify the <code>model</code> object to which you want add a variant.
<i>variantObj</i>	Variant object to create and add to the model object.
<i>NameValue</i>	Name of the variant object. <i>NameValue</i> is assigned to the <code>Name</code> property of the variant object.

### Description

`variantObj = addvariant(modelObj, 'NameValue')` creates a SimBiology variant object (*variantObj*) with the name *NameValue* and adds the variant object to the SimBiology model object *modelObj*. The variant object `Parent` property is assigned the value of *modelObj*.

A SimBiology variant object stores alternate values for properties on a SimBiology model. For more information on variants, see `Variant` object.

`variantObj2 = addvariant(modelObj, variantObj)` adds a SimBiology variant object (*variantObj*) to the SimBiology model object and returns another variant object *variantObj2*. The variant object *variantObj2* `Parent` property is assigned the value of *modelObj*. The `Active` property of *variantObj2* is set to `false` by default.

View properties for a variant object with the `get` command, and modify properties for a variant object with the `set` command.

**Note** Remember to use the `addcontent` method instead of using the `set` method on the `Content` property, because the `set` method replaces the data in the `Content` property, whereas `addcontent` appends the data.

---

To view the variants stored on a model object, use the `getvariant` method. To copy a variant object to another model, use `copyobj`. To remove a variant object from a SimBiology model, use the `delete` method.

## Examples

- 1 Create a model containing one species.

```
modelObj = sbiomodel('mymodel');  
compObj = addcompartment(modelObj, 'comp1');  
speciesObj = addspecies(compObj, 'A');
```

- 2 Add a variant object that varies the `InitialAmount` property of a species named A.

```
variantObj = addvariant(modelObj, 'v1');  
addcontent(variantObj, {'species', 'A', 'InitialAmount', 5});
```

## See Also

model object, variant object, `addcontent`, `commit`, `copyobj`, `delete`, `getvariant`

**Introduced in R2007b**

## boxplot(LeastSquaresResults, OptimResults, NLINResults)

Create box plot showing the variation of estimated SimBiology model parameters

### Syntax

```
boxplot(resultsObj)
```

### Description

`boxplot(resultsObj)` creates a box plot showing the variation of the estimated SimBiology model parameters.

### Input Arguments

#### **resultsObj** — Estimation results

`OptimResults` object | `NLINResults` object | vector of results objects

Estimation results, specified as an `OptimResults` object or `NLINResults` object, or vector of results objects which contains estimation results from running `sbiofit`.

### See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**



## boxplot(NLMEResults)

Create box plot showing the variation of estimated SimBiology model parameters

### Syntax

```
boxplot(resultsObj)
```

### Description

`boxplot(resultsObj)` creates a box plot showing the variation of the estimated SimBiology model parameters.

### Input Arguments

#### **resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results from running `sbiofitmixed`.

### See Also

NLMEResults object | sbiofitmixed

**Introduced in R2014a**

## ci2table

**Package:** SimBiology.fit

Return summary table of confidence interval results

## Syntax

```
tbl = ci2table(paraCI)
```

## Description

`tbl = ci2table(paraCI)` returns a summary table of confidence interval results from `paraCI`, a `ParameterConfidenceInterval` object or vector of objects.

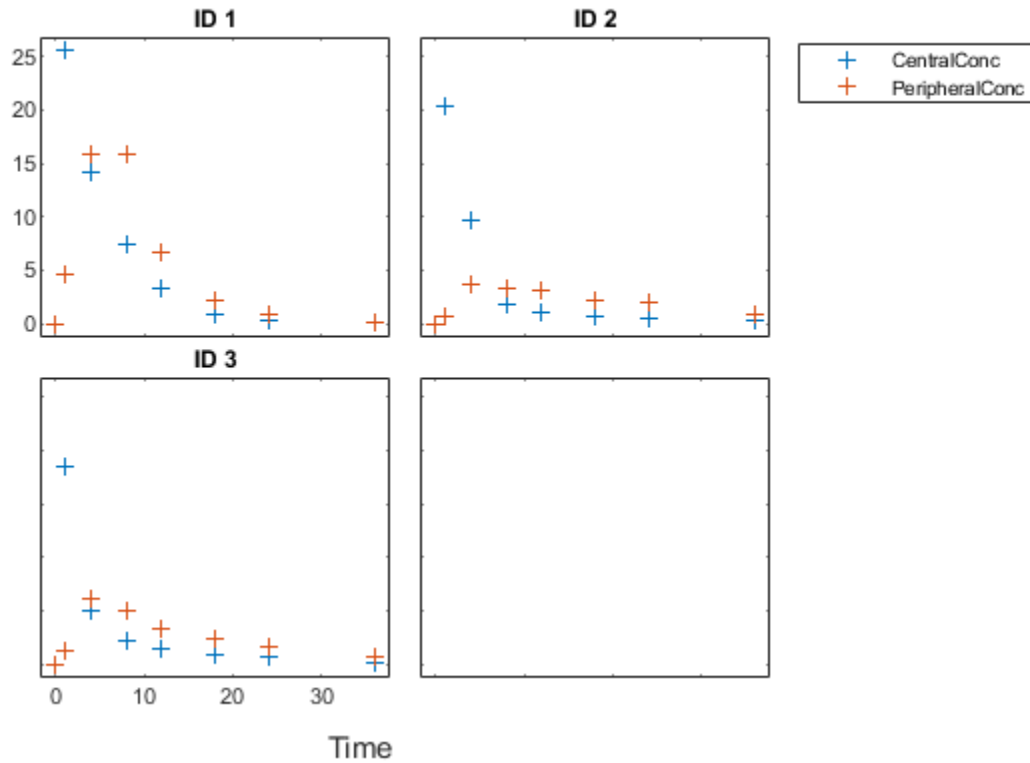
## Examples

### Compute Confidence Intervals for Estimated PK Parameters and Model Predictions

#### Load Data

Load the sample data to fit. The data is stored as a table with variables *ID*, *Time*, *CentralConc*, and *PeripheralConc*. This synthetic data represents the time course of plasma concentrations measured at eight different time points for both central and peripheral compartments after an infusion dose for three individuals.

```
clear all
load data10_32R.mat
gData = groupedData(data);
gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};
sbiotrellis(gData,'ID','Time',{'CentralConc','PeripheralConc'},'Marker','+',...
            'LineStyle','none');
```



## Create Model

Create a two-compartment model.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2 = addCompartment(pkmd, 'Peripheral');
model = construct(pkmd);
configset = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

### **Define Dosing**

Define the infusion dose.

```
dose                = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime     = 0;
dose.Amount        = 100;
dose.Rate          = 50;
dose.AmountUnits   = 'milligram';
dose.TimeUnits     = 'hour';
dose.RateUnits     = 'milligram/hour';
```

### **Define Parameters**

Define the parameters to estimate. Set the parameter bounds for each parameter. In addition to these explicit bounds, the parameter transformations (such as log, logit, or probit) impose implicit bounds.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam = estimatedInfo(paramsToEstimate, ...
                               'InitialValue', [1 1 1 1], ...
                               'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

### **Fit Model**

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

Perform a pooled fit, that is, one set of estimated parameters for all patients.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true);
```

### **Compute Confidence Intervals for Estimated Parameters**

Compute 95% confidence intervals for each estimated parameter in the unpooled fit.

```
ciParamUnpooled = sbioparameterci(unpooledFit);
```

### **Display Results**

Display the confidence intervals in a table format. For details about the meaning of each estimation status, see “Parameter Confidence Interval Estimation Status” on page 2-374.

```
ci2table(ciParamUnpooled)
```

```
ans =
```

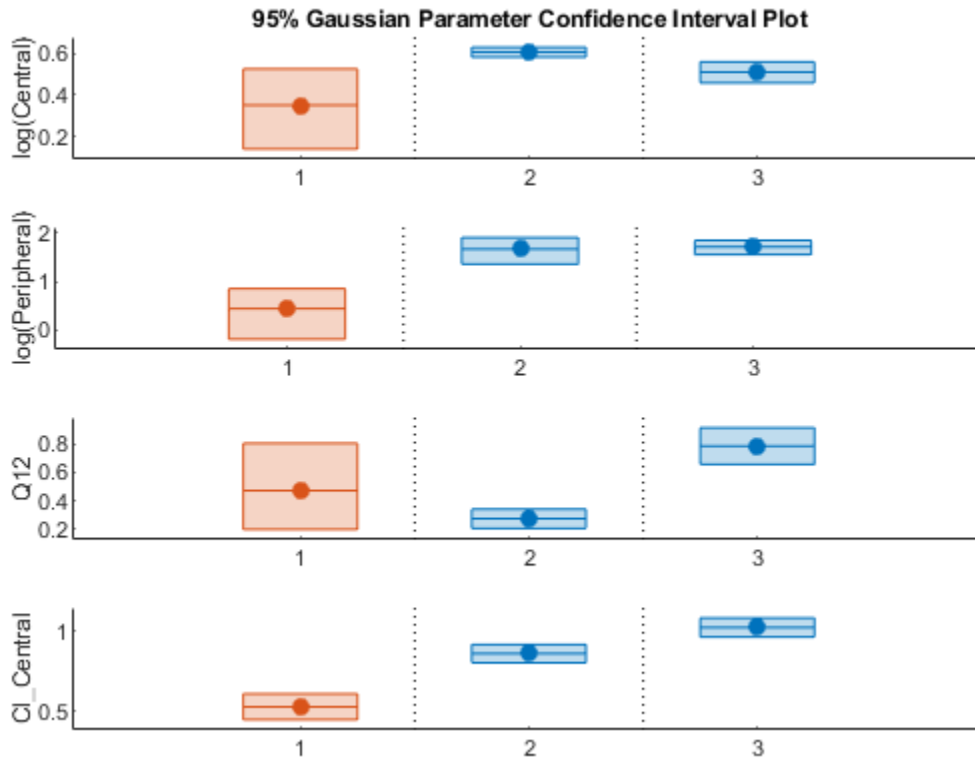
```
12x7 table
```

Group	Name	Estimate	ConfidenceInterval		Type	Alpha	
1	'Central'	1.422	1.1533	1.6906	Gaussian	0.05	est
1	'Peripheral'	1.5629	0.83143	2.3551	Gaussian	0.05	con
1	'Q12'	0.47159	0.20093	0.80247	Gaussian	0.05	con
1	'Cl_Central'	0.52898	0.44842	0.60955	Gaussian	0.05	est
2	'Central'	1.8322	1.7893	1.8751	Gaussian	0.05	su
2	'Peripheral'	5.3368	3.9133	6.7602	Gaussian	0.05	su
2	'Q12'	0.27641	0.2093	0.34351	Gaussian	0.05	su
2	'Cl_Central'	0.86034	0.80313	0.91755	Gaussian	0.05	su
3	'Central'	1.6657	1.5818	1.7497	Gaussian	0.05	su
3	'Peripheral'	5.5632	4.7557	6.3708	Gaussian	0.05	su
3	'Q12'	0.78361	0.65581	0.91142	Gaussian	0.05	su
3	'Cl_Central'	1.0233	0.96375	1.0828	Gaussian	0.05	su

Plot the confidence intervals. If the estimation status of a confidence interval is **success**, it is plotted in blue (the first default color). Otherwise, it is plotted in red (the second default color), which indicates that further investigation into the fitted parameters may be required. If the confidence interval is **not estimable**, then the function plots a red line with a centered cross. If there are any transformed parameters with estimated values 0 (for the log transform) and 1 or 0 (for the probit or logit transform), then no confidence intervals are plotted for those parameter estimates. To see the color order, type `get(groot, 'defaultAxesColorOrder')`.

Groups are displayed from left to right in the same order that they appear in the `GroupNames` property of the object, which is used to label the x-axis. The y-labels are the transformed parameter names.

```
plot(ciParamUnpooled)
```



Compute the confidence intervals for the pooled fit.

```
ciParamPooled = sbioparameterci(pooledFit);
```

Display the confidence intervals.

```
ci2table(ciParamPooled)
```

```
ans =
```

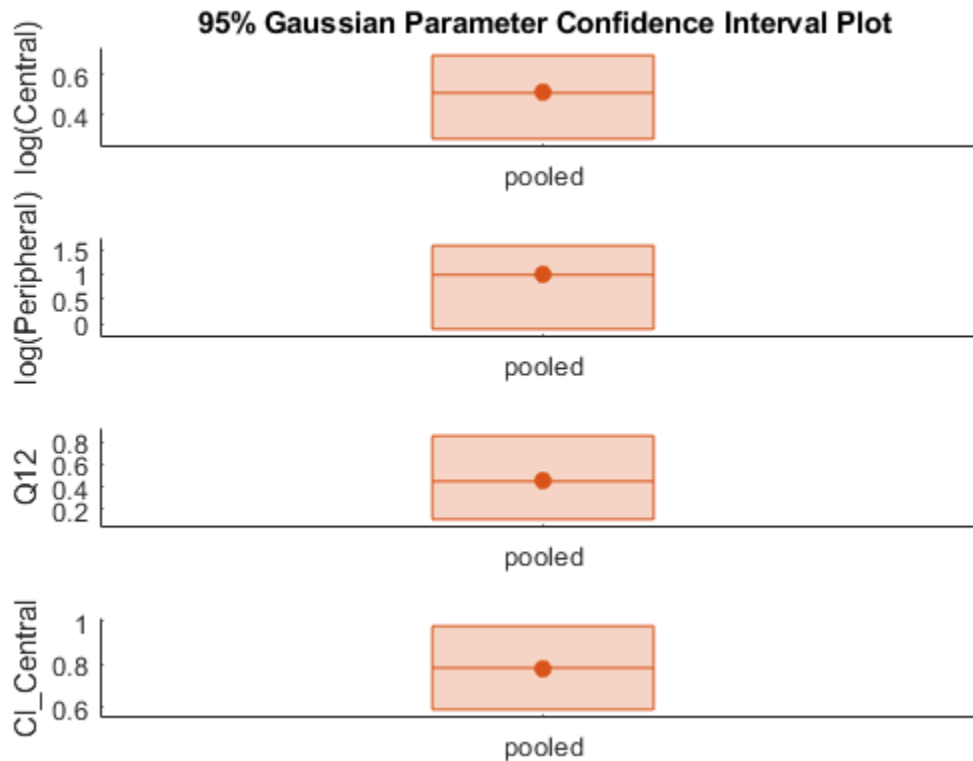
```
4x7 table
```

Group	Name	Estimate	ConfidenceInterval	Type	Alpha
-------	------	----------	--------------------	------	-------

pooled	'Central'	1.6626	1.3287	1.9965	Gaussian	0.05	es
pooled	'Peripheral'	2.687	0.89848	4.8323	Gaussian	0.05	co
pooled	'Q12'	0.44956	0.11445	0.85152	Gaussian	0.05	co
pooled	'Cl_Central'	0.78493	0.59222	0.97764	Gaussian	0.05	es

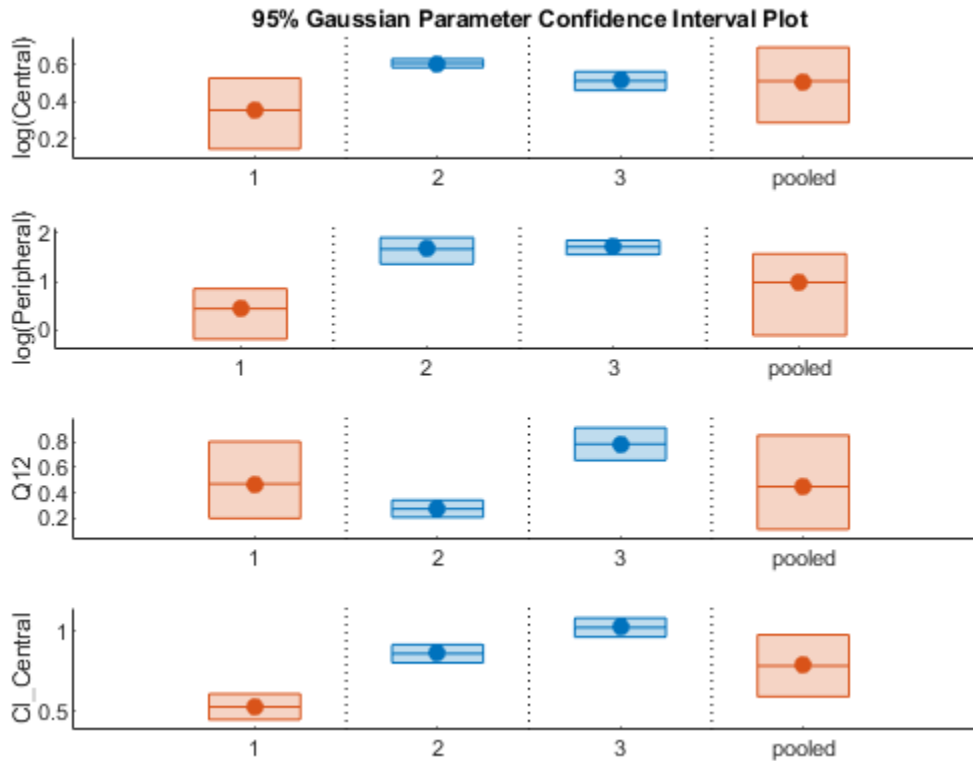
Plot the confidence intervals. The group name is labeled as "pooled" to indicate such fit.

```
plot(ciParamPooled)
```



Plot all the confidence interval results together. By default, the confidence interval for each parameter estimate is plotted on a separate axes. Vertical lines group confidence intervals of parameter estimates that were computed in a common fit.

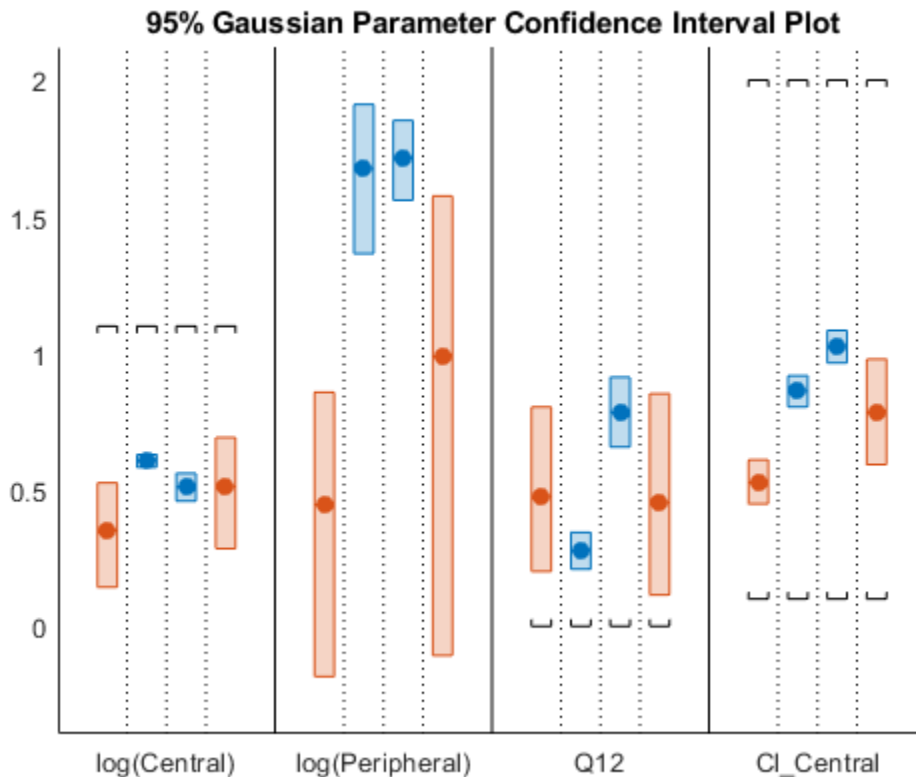
```
ciAll = [ciParamUnpooled;ciParamPooled];
plot(ciAll)
```



You can also plot all confidence intervals in one axes grouped by parameter estimates using the 'Grouped' layout.

```
plot(ciAll, 'Layout', 'Grouped')
```





In this layout, you can point to the center marker of each confidence interval to see the group name. Each estimated parameter is separated by a vertical black line. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets. Note the different scales on the y-axis due to parameter transformations. For instance, the y-axis of Q12 is in the linear scale, but that of Central is in the log scale due to its log transform.

### Compute Confidence Intervals for Model Predictions

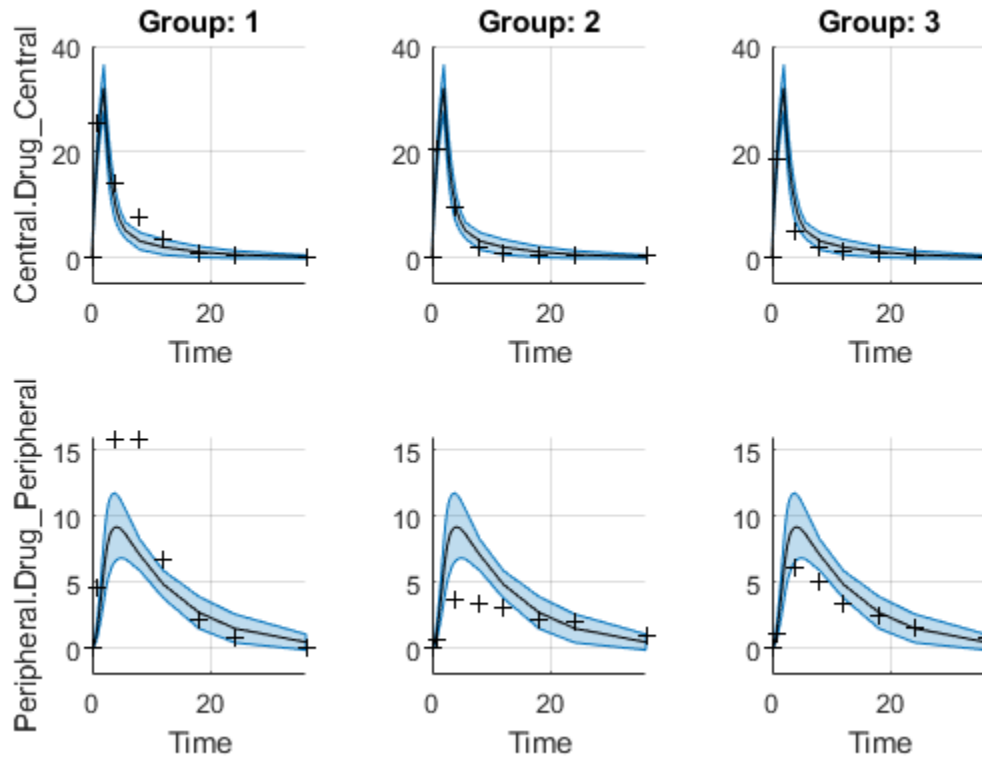
Calculate 95% confidence intervals for the model predictions, that is, simulation results using the estimated parameters.

```
% For the pooled fit
ciPredPooled = sbiopredictionci(pooledFit);
% For the unpooled fit
ciPredUnpooled = sbiopredictionci(unpooledFit);
```

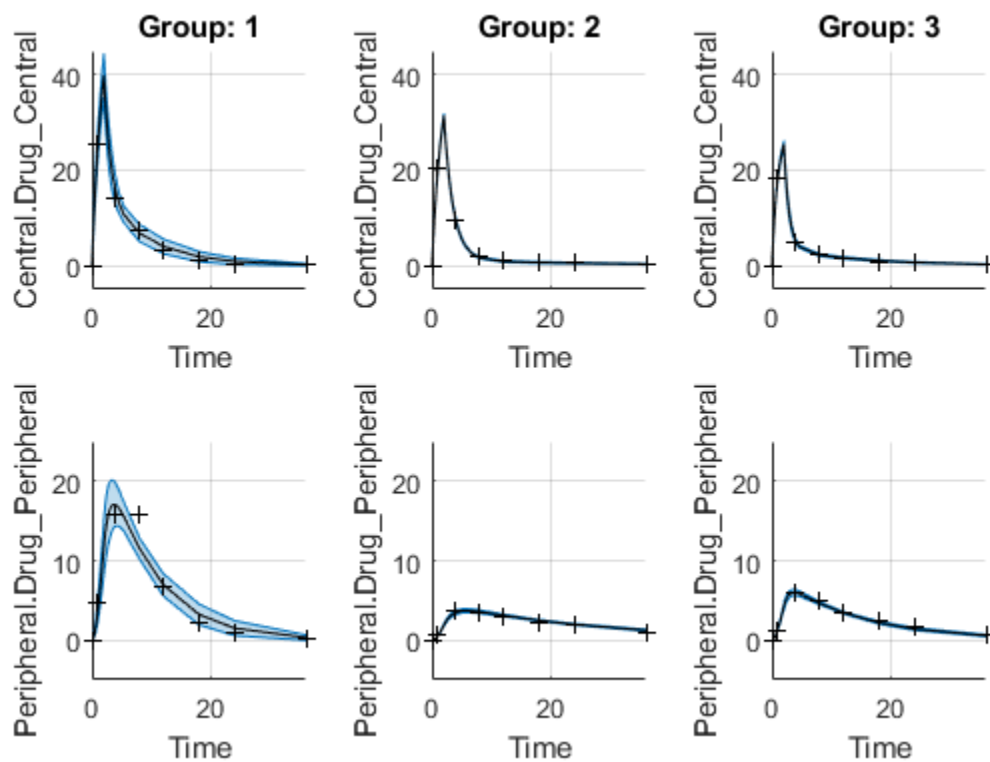
### Plot Confidence Intervals for Model Predictions

The confidence interval for each group is plotted in a separate column, and each response is plotted in a separate row. Confidence intervals limited by the bounds are plotted in red. Confidence intervals not limited by the bounds are plotted in blue.

```
plot(ciPredPooled)
```



```
plot(ciPredUnpooled)
```



## Input Arguments

### **paraCI** — Parameter confidence interval results

`ParameterConfidenceInterval` object | vector

Parameter confidence interval results, specified as a `ParameterConfidenceInterval` object or a vector of objects.

## Output Arguments

### **tbl** — Summary table for confidence interval results

Summary table for confidence interval results, returned as a `table`. The table contains the following columns.

<b>Column Name</b>	<b>Description</b>
<i>Group</i>	Group name
<i>Name</i>	Estimated parameter name
<i>Estimate</i>	Estimated parameter value
<i>ConfidenceInterval</i>	Confidence interval values
<i>Type</i>	Confidence interval type
<i>Alpha</i>	Confidence level
<i>Status</i>	Confidence interval estimation status (for details, see “Parameter Confidence Interval Estimation Status” on page 2-374)

## See Also

`ParameterConfidenceInterval` | `sbioparameterci`

**Introduced in R2017b**

## commit (variant)

Commit variant contents to model

### Syntax

```
commit(variantObj, modelObj)
```

### Arguments

<i>modelObj</i>	Specify the model object to which you want to commit a variant.
<i>variantObj</i>	Variant object to commit to the model object.

### Description

`commit(variantObj, modelObj)` commits the `Contents` property of a SimBiology variant object (*variantObj*) to the model object *modelObj*. The property values stored in the variant object replace the values stored in the model.

A SimBiology variant object stores alternate values for properties on a SimBiology model. For more information on variants, see `Variant` object.

The `Contents` are set on the model object in order of occurrence, with duplicate entries overwriting. If the `commit` method finds an incorrectly specified entry, an error occurs and the remaining properties defined in the `Contents` property are not set.

### Examples

- 1 Create a model containing one species.

```
modelObj = sbiomodel('mymodel');
compObj = addcompartment(modelObj, 'comp1');
speciesObj = addspecies(compObj, 'A', 10);
```

- 2 Add a variant object that varies the `InitialAmount` property of a species named A.

```
variantObj = addvariant(modelObj, 'v1');  
addcontent(variantObj, {'species', 'A', 'InitialAmount', 5});
```

- 3 Commit the contents of the variant (*variantObj*).

```
commit (variantObj, modelObj);
```

## See Also

`addvariant`, `Variant` object

**Introduced in R2007b**

# Compartment object

Options for compartments

## Description

The SimBiology compartment object represents a container for species in a model. Compartment size can vary or remain constant during a simulation. All models must have at least one compartment and all species in a model must be assigned to a compartment. Compartment names must be unique within a model.

Compartments allow you to define the size (*Capacity*) of physically isolated regions that may affect simulation, and associate pools of species within those regions. You can specify or change *Capacity* using rules, events, and variants, similar to species amounts or parameter values.

The model object stores compartments as a flat list. Each compartment stores information on its own organization; in other words a compartment has information on which compartment it lives within (*Owner*) and who it contains (*Compartments*).

The flat list of compartments in the model object lets you vary the way compartments are organized in your model without invalidating any expressions.

To add species that participate in reactions, add the reaction to the model using the `addreaction` method. When you define a reaction with a new species:

- If no compartment objects exist in the model, the `addreaction` method creates a compartment object (called '*unnamed*') in the model and adds the newly created species to that compartment.
- If only one compartment object exists in the model, the method creates a species object in that compartment.
- If there is more than one compartment object in the model, you must qualify the species name with the compartment name.

For example, `cell.glucose` denotes that you want to put the species named `glucose` into a compartment named `cell`. Additionally, if the compartment named `cell` does not exist, the process of adding the reaction creates the compartment and names it `cell`.

Alternatively, create and add a species object to a compartment object, using the `addspecies` method at the command line.

When you use the SimBiology desktop to create a new model, it adds an empty compartment (*unnamed*), to which you can add species.

You can specify reactions that cross compartments using the syntax *compartment1Name.species1Name -> compartment2Name.species2Name*. If you add a reaction that contains species from different compartments, and the reaction rate dimensions are concentration/time, all reactants should be from the same compartment.

In addition, if the reaction is reversible then there are two cases:

- If the kinetic law is `MassAction`, and the reaction rate dimensions are concentration/time, then the products must be from the same compartment.
- If the kinetic law is not `MassAction`, then both reactants and products must be in the same compartment.

See “Property Summary” on page 2-85 for links to compartment property reference pages. Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the graphical user interface.

## Constructor Summary

`addcompartment (model, compartment)`

Create compartment object

## Method Summary

Methods for compartment objects



addcompartment (model, compartment)	Create compartment object
addspecies (model, compartment)	Create species object and add to compartment object within model object
copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
findUsages (species, parameter, compartment)	Find out how a species, parameter, or compartment is used in a model
get (any object)	Get object properties
rename (compartment, parameter, species, reaction)	Rename object and update expressions
reorder (model, compartment, kinetic law)	Reorder component lists
set (any object)	Set object properties

## Property Summary

Properties for compartment objects

Capacity	Compartment capacity
CapacityUnits	Compartment capacity units
Compartments	Array of compartments in model or compartment
ConstantCapacity	Specify variable or constant compartment capacity
Name	Specify name of object
Notes	HTML text describing SimBiology object
Owner	Owning compartment
Parent	Indicate parent object
Species	Array of species in compartment object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

### See Also

AbstractKineticLaw object, Configset object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object

**Introduced in R2008a**

# ConfidenceInterval

Object containing confidence interval results

## Description

The `ConfidenceInterval` object is a superclass of two confidence interval objects: `PredictionConfidenceInterval` and `ParameterConfidenceInterval`. These objects contain confidence interval results computed with `sbiopredictionci` and `sbioparameterci`, respectively.

## Properties

### Type — Confidence interval type

'gaussian' | 'profilelikelihood' | 'bootstrap'

This property is read-only.

Confidence interval type, specified as 'gaussian', 'profileLikelihood' (for `ParameterConfidenceInterval` only), or 'bootstrap'

Example: 'bootstrap'

### GroupNames — Original group names from data used for fitting

cell array of character vectors

This property is read-only.

Original group names from the data used for fitting the model, specified as a cell array of character vectors. Each cell contains the name of a group.

Example: {'1'}{'2'}{'3'}

### Alpha — Confidence level

positive scalar

This property is read-only.

Confidence level,  $(1 - \text{Alpha}) * 100\%$ , specified as a positive scalar between 0 and 1.

Example: 0.01

### **Results — Confidence interval data**

table

This property is read-only.

Confidence interval data, specified as a table.

### **ExitFlags — Exit flags returned during calculation of bootstrap confidence intervals**

vector

This property is read-only.

Exit flags returned during the calculation of `bootstrap` confidence intervals only, specified as a vector of integers. Each integer is an exit flag returned by the estimation function (except `nlinfit`) used to fit parameters during bootstrapping. The same estimation function used in the original fit is used for bootstrapping.

Each flag indicates the success or failure status of the fitting performed to create a bootstrap sample. Refer to the reference page of the corresponding estimation function for the meaning of the exit flag.

If the estimation function does not return an exit flag, `ExitFlags` is set to `[]`. For the `gaussian` and `profileLikelihood` confidence intervals, `ExitFlags` is not supported and is always set to `[]`.

## **See Also**

`ParameterConfidenceInterval` | `PredictionConfidenceInterval` | `sbiparameterci` | `sbipredictionci`

**Introduced in R2017b**

# Configset object

Solver settings information for model simulation

## Description

The SimBiology configset object, also known as the configuration set object, contains the options that the solver uses during simulation of the model object. The configuration set object contains the following options for you to choose:

- Type of solver
- Stop time for the simulation
- Solver error tolerances, and for ode solvers — the maximum time step the solver should take
- Whether to perform sensitivity analysis during simulation
- Whether to perform dimensional analysis and unit conversion during simulation
- Species and parameter input factors for sensitivity analysis

A SimBiology model can contain multiple configsets with one being active at any given time. The active configset contains the settings that are used during the simulation. Use the method `setactiveconfigset` to define the active configset. Use the method `getConfigset` to return a list of configsets contained by a model. Use the method `addconfigset` to add a new configset to a model.

---

**Warning** The `Active` property of the `configset` object will be removed in a future release. Explicitly specify a `configset` object as an input argument when you simulate a model using `sbiosimulate`.

---

See “Property Summary” on page 2-90 for links to configset object property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the SimBiology desktop.

## Constructor Summary

addconfigset (model)     Create configuration set object and add to model object

## Method Summary

copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
set (any object)	Set object properties

## Property Summary

Active	Indicate object in use during simulation
AmountUnits	Amount unit used internally during simulation when UnitConversion is on
CompileOptions	Dimensional analysis and unit conversion options
MassUnits	Mass unit used internally during simulation when UnitConversion is on
MaximumNumberOfLogs	Maximum number of logs criteria to stop simulation
MaximumWallClock	Maximum elapsed wall clock time to stop simulation
Name	Specify name of object
Notes	HTML text describing SimBiology object
RuntimeOptions	Options for logged species
SensitivityAnalysisOptions	Specify sensitivity analysis options
SolverOptions	Specify model solver options
SolverType	Select solver type for simulation
StopTime	Simulation time criteria to stop simulation
TimeUnits	Show time units for dosing and simulation
Type	Display SimBiology object type

## See Also

AbstractKineticLaw object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object, Species object

**Introduced in R2006b**

## construct (PKModelDesign)

Construct SimBiology model from PKModelDesign object

### Syntax

```
[modelObj, pkModelMapObject] = construct(pkModelDesignObject)
[modelObj, pkModelMapObject, CovModelObj] =
construct(pkModelDesignObject)
```

### Arguments

<i>modelObj</i>	SimBiology model object specifying a pharmacokinetic model.
<i>pkModelMapObject</i>	Defines the roles of the components in <i>modelObj</i> . For details, see PKModelMap object.
<i>CovModelObj</i>	Defines the relationship between parameters and covariates. For details, see CovariateModel object.

### Description

[*modelObj*, *pkModelMapObject*] = `construct(pkModelDesignObject)` constructs a SimBiology model object, *modelObj*, containing the model components (such as compartments, species, reactions, and rules) required to represent the pharmacokinetic model specified in *pkModelDesignObject*. It also constructs *pkModelMapObject*, a PKModelMap object, which defines the roles of the model components.

The newly constructed model object, *modelObj*, is named 'Generated Model' (which you can change). It contains one compartment for each compartment specified in the PKCompartment property of *pkModelDesignObject*. Each compartment contains a species that represents a drug concentration. The compartments are connected with reversible reactions that models flux between compartments.



`[modelObj, pkModelMapObject, CovModelObj] = construct(pkModelDesignObject)` constructs `CovModelObj`, a `CovariateModel` object, which defines the relationship between parameters and covariates. Within the `Expression` property of `CovModelObj`, each parameter being estimated has an expression of the form `parameterName = exp(theta1 + eta1)` (without covariate dependencies), where `theta1` is a fixed effect, and `eta1` is a random effect. You can modify the expressions to add covariate dependencies. For details, see `CovariateModel` on page 2-98 object.

## See Also

`CovariateModel` object | `PKModelDesign` object | `PKModelMap` object

## Topics

“Create a Pharmacokinetic Model Using the Command Line”

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”

“Specify a Covariate Model”

**Introduced in R2009a**

## constructDefaultFixedEffectValues (covmodel)

Create initial estimate vector needed for fit

### Syntax

```
FEInitEstimates = constructDefaultFixedEffectValues(CovModelObj)
```

### Description

*FEInitEstimates* = `constructDefaultFixedEffectValues(CovModelObj)` creates *FEInitEstimates*, a structure containing the initial estimates for the fixed effects in *CovModelObj*, a `CovariateModel` object. These initial estimates are set to a default of zero, but you can edit these estimates. The number and names of the fields in the *FEInitEstimates* structure matches the number and names of fixed effects (theta values) in the Expression on page 3-85 property of *CovModelObj*.

---

**Tip** After creating the *FEInitEstimates* structure, you can edit it and use it to change the `FixedEffectValues` property of *CovModelObj*, before using the object as an input argument to `sbionlmeFit` or `sbionlmeFitSA`.

---

### See Also

`CovariateModel` on page 2-98 | Expression on page 3-85 | `FixedEffectValues` on page 3-94 | `sbionlmeFit` | `sbionlmeFitSA`

### Topics

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”  
“Specify a Covariate Model”

**Introduced in R2011b**

## copyobj (any object)

Copy SimBiology object and its children

### Syntax

```
copiedObj = copyobj(Obj, parentObj)
copiedObj = copyobj(modelObj)
```

### Arguments

<i>Obj</i>	Compartment, configuration set, event, kinetic law, model, parameter, reaction, rule, species, or variant object.	
	<b>Note</b> Abstract kinetic law objects cannot be copied using <code>copyobj</code> .	
<i>parentObj</i>	<b>If <i>copiedObj</i> is...</b>	<b><i>parentObj</i> must be...</b>
	configuration set, event, reaction, rule, or variant object	model object
	compartment object	compartment or model object
	species object	compartment object
	parameter object	model or kinetic law object
	kinetic law object	reaction object
	model object	<code>sbioroot</code>
<i>modelObj</i>	Model object to be copied.	
<i>copiedObj</i>	Output returned by the <code>copyobj</code> method with the parent set as specified in input argument ( <i>parentObj</i> ).	

## Description

*copiedObj* = `copyobj(Obj, parentObj)` makes a copy of a SimBiology object (*Obj*) and returns a pointer to the copy (*copiedObj*). In the copied object (*copiedObj*), this method assigns a value (*parentObj*) to the property `Parent`.

*copiedObj* = `copyobj(modelObj)` makes a copy of a model object (*modelObj*) and returns the copy (*copiedObj*). In the copied model object (*copiedObj*), this method assigns the root object to the property `Parent`.

---

**Note** When the `copyobj` method copies a model, it resets the `StatesToLog` property to the default value. Similarly, the `Inputs` and `Outputs` properties are not copied but rather left empty. Thus, when you simulate a copied model you see results for the default states, unless you manually update these properties.

---

## Examples

Create a reaction object separate from a model object, and then add it to a model.

- 1 Create a model object and add a reaction object.

```
modelObj1 = sbiomodel('cell');  
reactionObj = addreaction(modelObj1, 'a -> b');
```

- 2 Create a copy of the reaction object and assign it to another model object.

```
modelObj2 = sbiomodel('cell2');  
reactionObjCopy = copyobj(reactionObj, modelObj2);  
modelObj2.Reactions
```

```
SimBiology Reaction Array
```

```
Index:    Reaction:  
1         a -> b
```

## See Also

`sbiomodel`, `sbioroot`

**Introduced in R2006a**

## CovariateModel object

Define relationship between parameters and covariates

### Description

CovariateModel defines the relationship between estimated parameters and covariates.

---

**Tip** Use a CovariateModel object as an input argument to `sbiofitmixed` to fit a model with covariate dependencies. Before using the CovariateModel object, set the FixedEffectValues on page 3-94 property to specify the initial estimates for the fixed effects.

---

### Construction

`CovModelObj = CovariateModel` creates an empty CovariateModel object.

`CovModelObj = CovariateModel(Expression)` creates a CovariateModel object with its Expression property set to *Expression*, a character vector, string, string vector, or cell array of character vectors, where each character vector or string represents the relationship between a parameter being estimated and one or more covariates. *Expression* must denote fixed effects with the prefix *theta*, and random effects with the prefix *eta*. Each character vector or string in *Expression* must be in the form:

```
parameterName = relationship
```

This example of an expression defines the relationship between a parameter (`volume`) and a covariate (`weight`), with fixed effects, but no random effects:

```
Expression = {'volume = theta1 + theta2*weight'};
```

This table illustrates expression formats for some common parameter-covariate relationships.

Parameter-Covariate Relationship	Expression Format
Linear with random effect	$Cl = \theta_1 + \theta_2 * WEIGHT + \epsilon_1$
Exponential without random effect	$Cl = \exp(\theta_{Cl} + \theta_{Cl\_WT} * WEIGHT)$
Exponential, WEIGHT centered by mean, has random effect	$Cl = \exp(\theta_1 + \theta_2 * (WEIGHT - \text{mean}(WEIGHT)) + \epsilon_1)$
Exponential, log(WEIGHT), which is equivalent to power model	$Cl = \exp(\theta_1 + \theta_2 * \log(WEIGHT) + \epsilon_1)$
Exponential, dependent on WEIGHT and AGE, has random effect	$Cl = \exp(\theta_1 + \theta_2 * WEIGHT + \theta_3 * AGE + \epsilon_1)$
Inverse of probit, dependent on WEIGHT and AGE, has random effect	$Cl = \text{probitinv}(\theta_1 + \theta_2 * WEIGHT + \theta_3 * AGE + \epsilon_1)$
Inverse of logit, dependent on WEIGHT and AGE, has random effect	$Cl = \text{logitinv}(\theta_1 + \theta_2 * WEIGHT + \theta_3 * AGE + \epsilon_1)$

**Tip** To simultaneously fit data from multiple dose levels, use a `CovariateModel` object as an input argument to `sbiofitmixed`, and omit the random effect (`eta`) from the `Expression` property in the `CovariateModel` object.

## Method Summary

`constructDefaultFixedEffectValues`  
(`covmodel`)

Create initial estimate vector needed for fit

`verify` (`covmodel`)

Check covariate model for errors

## Property Summary

CovariateLabels (CovariateModel)	Labels for covariates in <code>CovariateModel</code> object
Expression (CovariateModel)	Define relationship between parameters and covariates
FixedEffectDescription (CovariateModel)	Descriptions of fixed effects in <code>CovariateModel</code> object
FixedEffectNames (CovariateModel)	Names of fixed effects in <code>CovariateModel</code> object
FixedEffectValues (CovariateModel)	Values for initial estimates of fixed effects in <code>CovariateModel</code> object
ParameterNames (CovariateModel)	Names of parameters in <code>CovariateModel</code> object
RandomEffectNames (CovariateModel)	Names of random effects in <code>CovariateModel</code> object

## Examples

Create a `CovariateModel` object and set the `Expression` property to define the relationship between two parameters (clearance and volume) and two covariates (weight and age) using fixed effects (`thetas`) and random effects (`etas`):

```
covModelObj = CovariateModel  
covModelObj.Expression = {'CL = theta1 + theta2*WT + eta1', 'V = theta3 + theta4*AGE +
```

## See Also

`PKData` object | `PKModelDesign` object | `construct` | `getCovariateData` on page 2-201 | `sbionlmefit` | `sbionlmefitsa`

## Topics

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”  
“Specify a Covariate Model”



**Introduced in R2011b**

## covariateModel(NLMEResults)

Return a copy of the covariate model that was used for the nonlinear mixed-effects estimation using `sbiofitmixed`

### Syntax

```
covmodel = covariateModel(resultsObj)
```

### Description

`covmodel = covariateModel(resultsObj)` returns a copy of the covariate model that was used for the nonlinear mixed-effects estimation using `sbiofitmixed`.

### Input Arguments

#### **resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results from running `sbiofitmixed`.

### Output Arguments

#### **covmodel** — Covariate model

CovariateModel object

Covariate model, returned as a CovariateModel object, that was used for the nonlinear mixed-effects estimation using `sbiofitmixed`. The model describes the relationship between SimBiology model parameters, fixed effects, random effects, and covariates.

## See Also

NLMEResults object | sbiofitmixed

**Introduced in R2014a**

# createDoses

Create dose objects from groupedData object

## Syntax

```
doseArray = createDoses(grpData, amountVarNames)
doseArray = createDoses(grpData, amountVarNames, rateVarNames)
doseArray = createDoses(grpData, amountVarNames, rateVarNames,
tempDoses)
doseArray = createDoses(grpData, amountVarNames, rateVarNames,
tempDoses, groups)
```

## Description

`doseArray = createDoses(grpData, amountVarNames)` creates an array of SimBiology dose objects using dose times and amount data specified in `grpData`, with one row per group and one column per dose amount variable.

`grpData.Properties.IndependentVariable` specifies which variable contains dose times, and `amountVarNames` specifies which variables contain valid dose amounts.

`doseArray = createDoses(grpData, amountVarNames, rateVarNames)` uses dose rate variables specified by `rateVarNames`.

`doseArray = createDoses(grpData, amountVarNames, rateVarNames, tempDoses)` uses template doses specified by `tempDoses` as templates for dose objects in `doseArray`. In other words, this argument lets you copy some of the template dose properties, such as `TargetName`, `DurationParameterName`, and `LagParameterName`, to dose objects in `doseArray`.

`doseArray = createDoses(grpData, amountVarNames, rateVarNames, tempDoses, groups)` specifies which groups in `grpData` to create doses for.

## Examples

## Create Array of Doses from groupedData

Load the sample data set.

```
load pheno.mat ds
```

Create a groupedData object from the data set ds.

```
grpData = groupedData(ds);
```

Display the object properties.

```
grpData.Properties
```

```
ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Observations' 'Variables'}
    VariableNames: {'ID' 'TIME' 'DOSE' 'WEIGHT' 'APGAR' 'CONC'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
    RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: 'ID'
    IndependentVariableName: 'TIME'
```

GroupVariableName and IndependentVariableName have been automatically assigned to 'ID' and 'Time', respectively.

Create an array of dose objects using the dosing information from the groupedData specified by the DOSE variable. Each row (dose object) represents a dosing schedule for each individual (group).

```
doseArray = createDoses(grpData, 'DOSE');
```

## Input Arguments

### grpData — Grouped data

groupedData object

Grouped data, specified as a groupedData object.

Set `grpData.Properties.IndependentVariable` to a valid variable in `grpData` that specifies the dose times. `grpData.Properties.GroupVariableName` optionally identifies a grouping variable. `grpData.Properties.VariableUnits` optionally specifies units for the corresponding variables. If the specified units are invalid, a warning is issued, and the units of corresponding doses in `doseArray` are set to empty character vectors ( ' ' ).

### **amountVarNames — Amount variable names**

character vector | string | cell array of character vectors | string vector

Amount variable names, specified as a character vector, string, string vector, or cell array of character vectors that specifies variables in `grpData` that define dose amounts. Each character vector or string must specify a valid amount variable.

An amount variable is valid if it is a real, nonnegative column vector containing no infinite values.

### **rateVarNames — Rate variable names**

character vector | string | cell array of character vectors | string vector

Rate variable names, specified as a character vector, string, string vector, or cell array of character vectors that specify variables in `grpData` that define dose rates. If it is empty [ ] or {}, it indicates that there are no dose rates. If it is not empty, it must be a character vector, string, cell array of character vectors or string vector of names of the same length as `amountVarNames`. Individual names can be empty ( ' ' or "" ) to indicate no dose rates for the corresponding doses or can be valid variable names in `grpData` specifying dose rates.

A rate variable is valid if it is a real, nonnegative column vector containing no infinite values. NaN rate values are allowed, but they are treated the same as the rate values of 0, that is, the doses are treated as bolus (instantaneous) doses.

### **tempDoses — Template doses**

dose object | vector of dose object | [ ]

Template doses, specified as a dose object (`ScheduleDose` object or `RepeatDose` object), vector of dose objects, or empty array [ ] .

Use this argument to copy the following template dose properties to each dose in `doseArray`: `TargetName`, `DurationParameterName`, `LagParameterName`, `Notes`, `Tag`, and `UserData`.

If `tempDoses` is a single dose object, these properties from the object are copied to all doses in `doseArray`. If it is a vector, it must have the same length as `amountVarNames`, and these properties from each element (dose) are copied to the corresponding column of doses in `doseArray`.

The `Name` property of each dose in `doseArray` consists of the `Name` of the template dose followed by the group name in parentheses, such as `'DailyDose (Patient1)'`.

If you do not specify units in `grpData.Properties.VariableUnits`, the following template dose units properties are copied to doses in `doseArray`: `AmountUnits`, `RateUnits`, and `TimeUnits`.

### **groups — Group names**

`[]` (default) | character vector | string vector | vector

Group names, specified as a character vector, string vector, an empty array `[]`, or a vector of data types that can be converted to a categorical vector. For a list of supported data types, see `categorical`.

By default, `groups` is set to `[]`, meaning the function creates doses for each group in `grpData`, with `doseArray` containing one row per group in `grpData`, in the order of the first occurrence of each group in `grpData`.

If you specify any group, the function converts those groups and the grouping variable in `grpData` to categorical vectors and compares them. The *i*th row of `doseArray` corresponds to the *i*th group specified in `groups`.

## **Output Arguments**

### **doseArray — SimBiology dose objects**

2-D matrix of dose objects

SimBiology dose objects, returned as a 2-D matrix of dose objects containing dose time and amount data from `grpData`. If dose times for a particular dose in `grpData` are regularly spaced, then the corresponding dose object in `doseArray` is a `RepeatDose` object. Otherwise, the corresponding dose object is a `ScheduleDose` object.

---

**Note** If there is a single dose time, then the dose object is represented as a `ScheduleDose` object.

---

## **See Also**

RepeatDose object | ScheduleDose object | groupedData | table

**Introduced in R2014a**



## createSimFunction (model)

Create SimFunction object

### Syntax

```
F = createSimFunction(model,params,observables,dosed)
F = createSimFunction(model,params,observables,dosed, variants)
F = createSimFunction( ____,Name,Value)
```

### Description

`F = createSimFunction(model,params,observables,dosed)` creates a `SimFunction` object `F` that you can execute like a function handle. The `params` and `observables` arguments define the inputs and outputs of the function `F` when it is executed, and `dosed` defines the dosing information of species. See `SimFunction` object for details on how to execute `F`.

`F = createSimFunction(model,params,observables,dosed, variants)` creates a `SimFunction` object, applying the values stored in `variants`, a vector of variant objects, as the model baseline values.

`F = createSimFunction( ____,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

---

#### Note

- Active doses and variants of the model are ignored when `F` is executed.
  - `F` is immutable after it is created.
  - `F` is automatically accelerated at the first function execution. However, manually accelerate the object if you want it accelerated in your deployment applications.
-

## Input Arguments

### **model** — SimBiology model

SimBiology model object

SimBiology model, specified as a `SimBiology model` object.

### **params** — Inputs of SimFunction F

character vector | cell array of character vectors | {}

Inputs of `SimFunction F`, specified as a character vector, cell array of character vectors, or an empty cell array {}. The character vectors represent the names of model quantities (species, compartments, or parameters) that define the inputs of `F`. Use an empty cell array {} to create a `SimFunction` object that has no parameters.

To unambiguously name a model quantity, use the qualified name, which includes the name of the compartment. To name a reaction-scoped parameter, use the reaction name to qualify the parameter. If the name is not a valid MATLAB variable name, surround it by square brackets such as `[reaction 1].[parameter 1]`.

### **observables** — Outputs of SimFunction F

character vector | cell array of character vectors

Outputs of `SimFunction F`, specified as a character vector or cell array of character vectors. The character vectors represent the names of model quantities (species, compartments, or parameters) that define the outputs of `F`.

### **dosed** — Dosed species or dose objects

character vector | cell array of character vectors | vector of dose objects | []

Dosed species or dose objects, specified as a character vector, cell array of character vectors, vector of dose objects, or empty array []. Use [] to specify no species are dosed during simulation.

If it is a cell array of character vectors, it must be 1-by- $N$  array, where  $N$  is the number of dosed species names. You can use duplicate species names if you plan to use multiple doses on page 2-0 for the same species when you run the `SimFunction F`. Using only dosed species names contains no information on the dose properties. If you have a dose object that contains parameterized properties such as `Amount`, use the dose object as input instead of just species names to transfer such parameter information to the created `SimFunction F`.

If it is a vector of dose objects, it must be 1-by- $N$  vector, where  $N$  is the number of dose objects. If dose objects have properties with nondefault numeric values, these values are ignored and a warning is issued. Only `TargetName`, `DurationParameterName`, `LagParameterName`, and parameterized properties are used to create the `SimFunction` object `F`, that is, to define the `Dosed` property of `F`. For details on how the `Dosed` property table is populated, see “Property Summary” on page 2-465.

The number of dosed species defined here, that is, the `Dosed` property of the created `SimFunction` object `F`, must be consistent with the dose input argument `u` on page 2-0 of the `SimFunction` `F` when you execute the object. In other words, the number of elements ( $N$ ) in the `Dosed` property of `F` must be equal to the number of elements (columns) in the cell array `u`. The order of dosed species in `Dosed` must also match the order of dose tables in `u`.

### variants — Alternate model values

variant object | vector of variant objects

Alternate model values, specified as a variant or vector of variant objects. These values are applied as the model baseline values when the `SimFunction` object is created. If there are multiple variants referring to the same model element, the last occurrence is used.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'UseParallel', true` specifies to execute the `SimFunction` `F` in parallel.

### UseParallel — Flag to execute SimFunction F in parallel

false (default) | true

Flag to execute `SimFunction` `F` in parallel, specified as the comma-separated pair consisting of `'UseParallel'` and `true` or `false`. If `true` and Parallel Computing Toolbox is available, the `SimFunction` `F` is executed in parallel.

Example: `'UseParallel', true`

### SensitivityOutputs — Sensitivity output factors

{ } (default) | cell array of character vectors | 'all'

Sensitivity output factors, specified as the comma-separated pair consisting of 'SensitivityOutputs' and a cell array of character vectors. The character vectors are the names of model quantities (species and parameters) for which you want to compute the sensitivities. The default is {} meaning there is no output factors. Output factors are the numerators of time-dependent derivatives explained in “Sensitivity Calculation”.

Use the keyword 'all' to specify all model quantities listed in the `observables` argument as sensitivity outputs. However, {'all'} means a model quantity named `all` in the model.

You must specify both 'SensitivityOutputs' and 'SensitivityInputs' name-value pair arguments for sensitivity calculations.

Example: 'SensitivityOutputs','all'

### **SensitivityInputs — Sensitivity input factors**

{ } (default) | cell array of character vectors | 'all'

Sensitivity input factors, specified as the comma-separated pair consisting of 'SensitivityInputs' and a cell array of character vectors. The character vectors are the names of model quantities (species, compartments, and parameters) with respect to which you want to compute the sensitivities. The default is {} meaning no input factors. Input factors are the denominators of time-dependent derivatives explained in “Sensitivity Calculation”.

Use the keyword 'all' to specify all model quantities listed in the `params` argument as sensitivity inputs. However, {'all'} means a model quantity named `all` in the model.

You must specify both 'SensitivityOutputs' and 'SensitivityInputs' name-value pair arguments for sensitivity calculations.

Example: 'SensitivityInputs',{'Reaction1.c1','Reaction1.c2'}

### **SensitivityNormalization — Normalization for calculated sensitivities**

'None' (default) | 'Half' | 'Full'

Normalization for calculated sensitivities, specified as the comma-separated pair consisting of 'SensitivityNormalization' and 'None', 'Half', or 'Full'.

- 'None' — No normalization (default)
- 'Half' — Normalization relative to the numerator only
- 'Full' — Full dedimensionalization

For details, see Normalization.

Example: 'SensitivityNormalization','Full'

## Output Arguments

### F — SimFunction

SimFunction object | SimFunctionSensitivity object

SimFunction, returned as a SimFunction object or SimFunctionSensitivity object. You can execute F like a function handle.

F is a SimFunctionSensitivity object if you specify non-empty 'SensitivityOutputs' and 'SensitivityInputs' name-value pair arguments.

## Examples

### Create a SimFunction Object

This example uses a radioactive decay model with the first-order reaction  $\frac{dz}{dt} = c \cdot x$ , where x and z are species and c is the forward rate constant.

Load the sample project containing the radioactive decay model m1.

```
sbioloadproject radiodecay;
```

Create a SimFunction object, specifying the parameter Reaction1.c to be scanned, and species x as the output of the function with no dosed species.

```
f = createSimFunction(m1, 'Reaction1.c','x', [])
```

```
f =  
SimFunction
```

Parameters:

Name	Value	Type	Units
_____	_____	_____	_____

```
'Reaction1.c'    0.5    'parameter'    '1/second'
```

Observables:

Name	Type	Units
'x'	'species'	'molecule'

Dosed: None

If the `UnitConversion` option was set to `false` when the `SimFunction` object `f` was created, the table does not display the units of the model quantities.

To illustrate this, first set the `UnitConversion` option to `false`.

```
cs = getconfigset(m1);
cs.CompileOptions.UnitConversion = false;
```

Create the `SimFunction` object as before and note that the variable named `Units` disappears.

```
f = createSimFunction(m1, {'Reaction1.c'}, {'x'}, [])
```

```
f =
SimFunction
```

Parameters:

Name	Value	Type
'Reaction1.c'	0.5	'parameter'

Observables:

Name	Type
'x'	'species'

Dosed: None

If any of the species in the model is being dosed, specify the names of dosed species as the last argument. For example, if the species `x` is being dosed, specify it as the last argument.

```
f = createSimFunction(m1, {'Reaction1.c'}, {'x'}, 'x')
```

```
f =  
SimFunction
```

Parameters:

Name	Value	Type
'Reaction1.c'	0.5	'parameter'

Observables:

Name	Type
'x'	'species'

Dosed:

TargetName
'x'

Once the `SimFunction` object is created, you can execute it like a function handle and perform parameter scans (in parallel if Parallel Computing Toolbox™ is available), Monte Carlo simulations, and scans with multiple or vectorized doses. See `SimFunction` object for more examples.

### Create a SimFunction Object with Dosing Information

This example creates a `SimFunction` object with dosing information using a `RepeatDose` or `ScheduleDose` object or a vector of these objects. However, if any dose object contains data such as `StartTime`, `Amount`, and `Rate`, such data are ignored, and

a warning is issued. Only data, if available, used are TargetName, LagParameterName, and DurationParameterName of the dose object.

Load the sample project containing the radioactive decay model m1.

```
sbioloadproject radiodecay;
```

Create a RepeatDose object and specify its properties.

```
rdose = sbiodose('rd');
rdose.TargetName = 'x';
rdose.StartTime = 5;
rdose.TimeUnits = 'second';
rdose.Amount = 300;
rdose.AmountUnits = 'molecule';
rdose.Rate = 1;
rdose.RateUnits = 'molecule/second';
rdose.Interval = 100;
rdose.RepeatCount = 2;
```

Add a lag parameter and duration parameter to the model.

```
lagPara = addparameter(m1, 'lp');
lagPara.Value = 1;
lagPara.ValueUnits = 'second';
duraPara = addparameter(m1, 'dp');
duraPara.Value = 1;
duraPara.ValueUnits = 'second';
```

Set these parameters to the dose object.

```
rdose.LagParameterName = 'lp';
rdose.DurationParameterName = 'dp';
```

Create a SimFunction object f using the RepeatDose object rdose that you just created.

```
f = createSimFunction(m1, {'Reaction1.c'}, {'x', 'z'}, rdose)
```

```
Warning: Some Dose objects in DOSED had data. This data
will be ignored.
```

```
> In SimFunction>SimFunction.SimFunction at 847
  In SimFunction>SimFunction.createSimFunction at 374
```

```
f =
```



SimFunction

Parameters:

Name	Value	Type	Units
'Reaction1.c'	0.5	'parameter'	'1/second'

Observables:

Name	Type	Units
'x'	'species'	'molecule'
'z'	'species'	'molecule'

Dosed:

TargetName	TargetDimension
'x'	'Amount(e.g. mole or molecule)'

DurationParameterName	DurationParameterValue
'dp'	1

DurationParameterUnits	LagParameterName
'second'	'lp'

LagParameterValue	LagParameterUnits
1	'second'

A warning message appears because the `rdose` object contains data (`StartTime`, `Amount`, `Rate`) that are ignored by the `createSimFunction` method.

### Scan Parameters of the Lotka-Volterra Model

This example shows how to execute different signatures of the `SimFunction` object to simulate and scan parameters of the Lotka-Volterra (predator-prey) model described by Gillespie [1].

Load the sample project containing the model `m1`.

```
sbioloadproject lotka;
```

Create a `SimFunction` object `f` with `c1` and `c2` as input parameters to be scanned, and `y1` and `y2` as the output of the function with no dosed species.

```
f = createSimFunction(m1,{'Reaction1.c1', 'Reaction2.c2'},{'y1', 'y2'}, [])
```

```
f =  
SimFunction
```

Parameters:

Name	Value	Type
'Reaction1.c1'	10	'parameter'
'Reaction2.c2'	0.01	'parameter'

Observables:

Name	Type
'y1'	'species'
'y2'	'species'

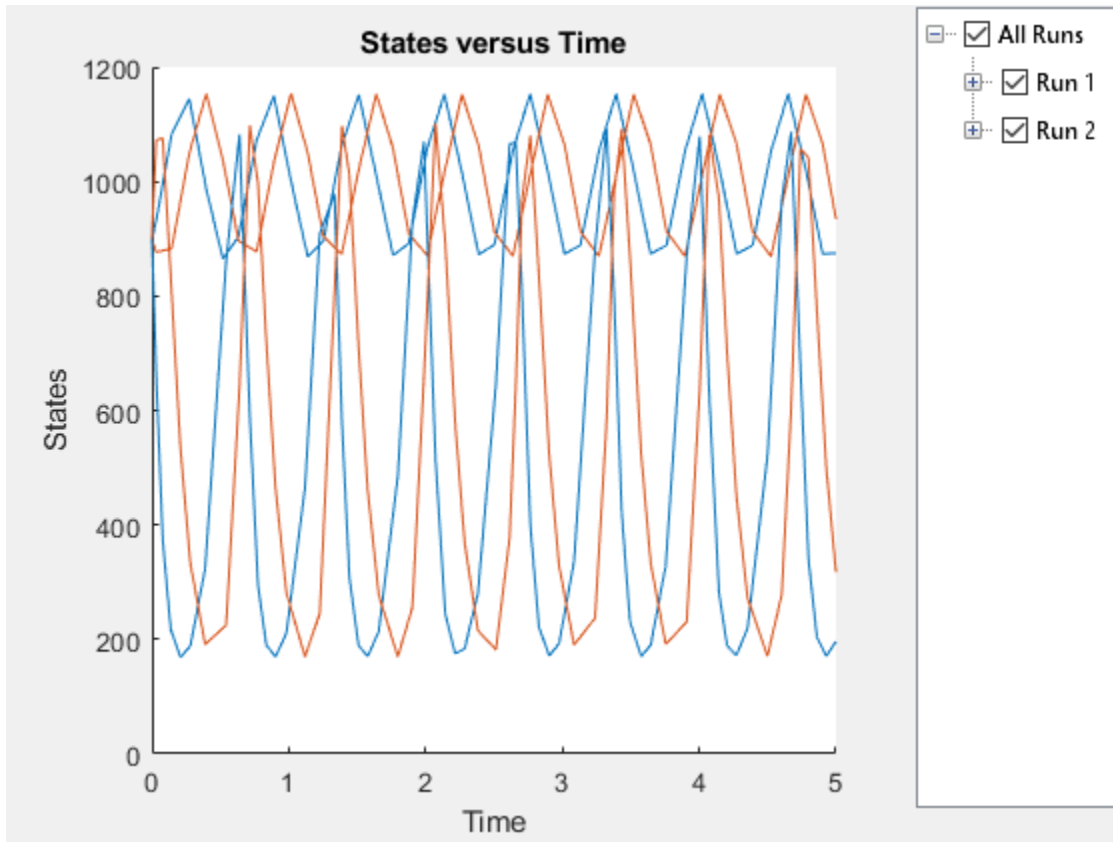
Dosed: None

Define an input matrix that contains values for each parameter (`c1` and `c2`) for each simulation. The number of rows indicates the total number of simulations, and each simulation uses the parameter values specified in each row.

```
phi = [10 0.01; 10 0.02];
```

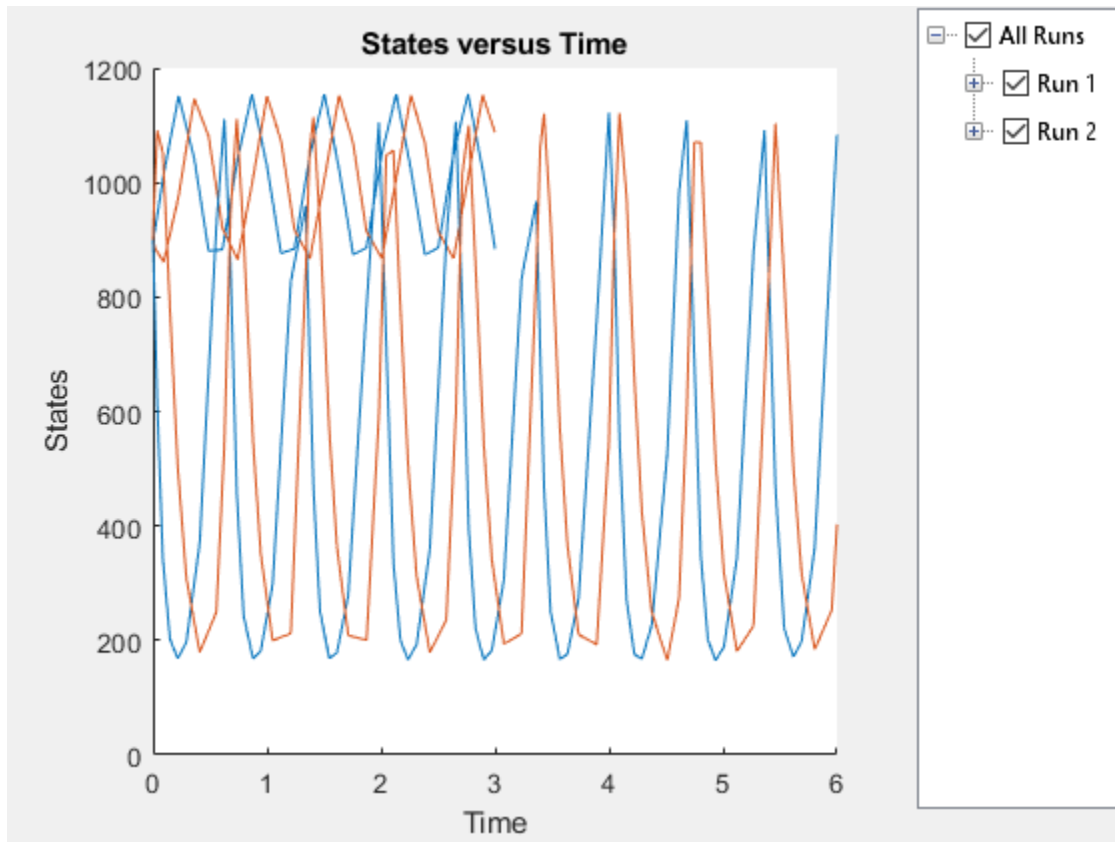
Run simulations until the stop time is 5 and plot the simulation results.

```
sbioplot(f(phi, 5));
```



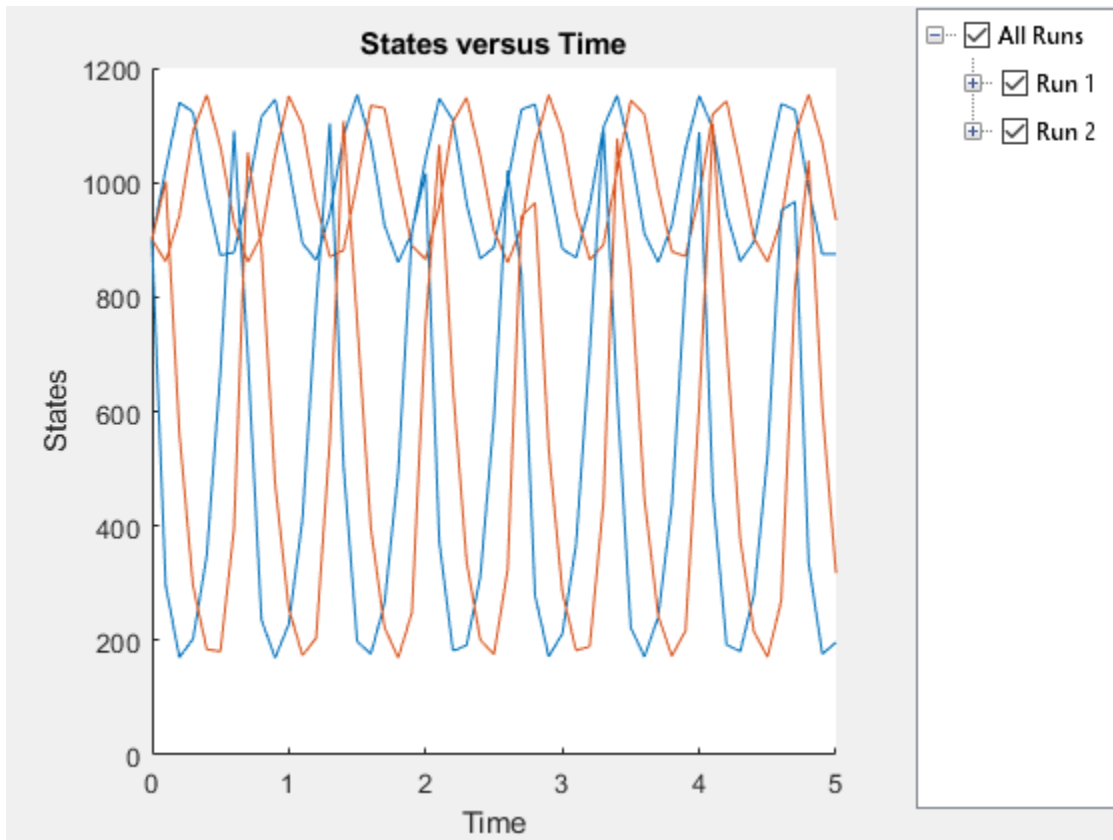
You can also specify a vector of different stop times for each simulation.

```
t_stop = [3;6];  
sbioplot(f(phi, t_stop));
```



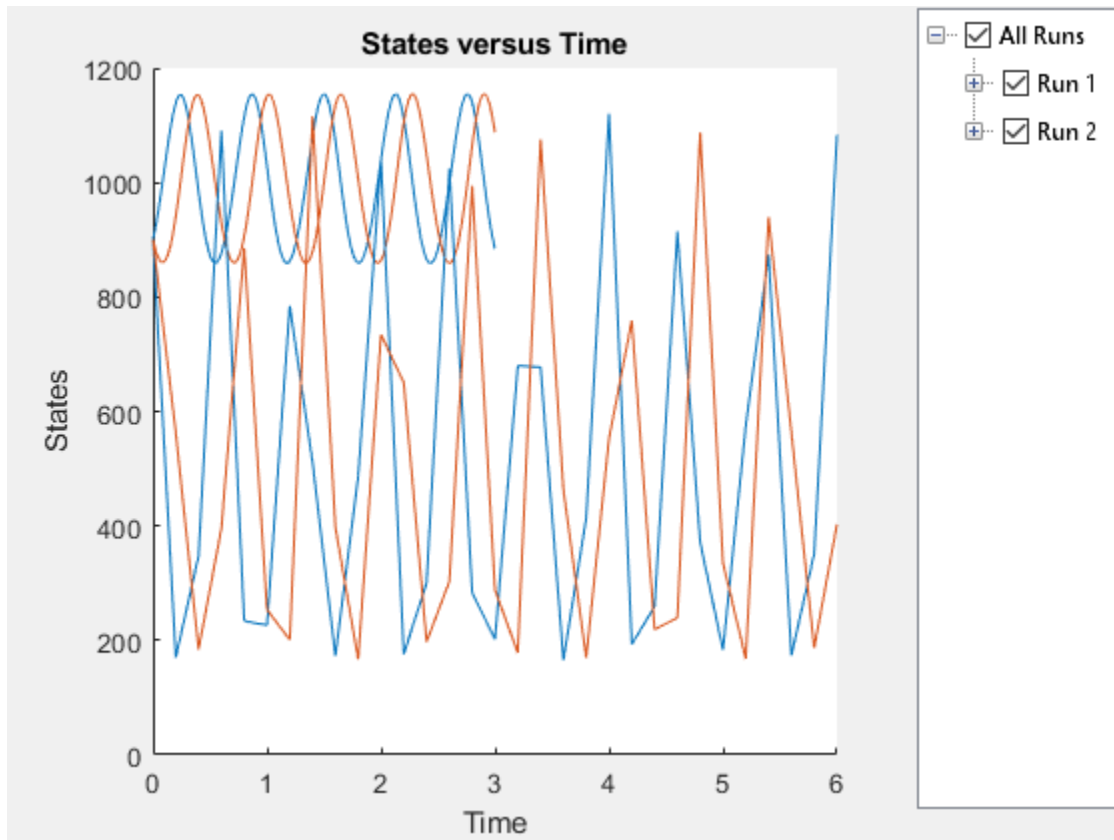
Next, specify the output times as a vector.

```
t_output = 0:0.1:5;  
sbioplot(f(phi,[],[],t_output));
```



Specify output times as a cell array of vectors.

```
t_output = {0:0.01:3, 0:0.2:6};  
sbioplot(f(phi, [], []), t_output);
```



### Calculate Sensitivities Using SimFunctionSensitivity Object

This example shows how to calculate sensitivities of some species in the Lotka-Volterra model using the SimFunctionSensitivity object.

Load the sample project.

```
sbioloadproject lotka;
```

Define the input parameters.

```
params = {'Reaction1.c1', 'Reaction2.c2'};
```

Define the observed species, which are the outputs of simulation.

```
observables = {'y1', 'y2'};
```

Create a `SimFunctionSensitivity` object. Set the sensitivity output factors to all species (y1 and y2) specified in the `observables` argument and input factors to those in the `params` argument (c1 and c2) by using the keyword 'all'.

```
f = createSimFunction(m1,params,observables,[], 'SensitivityOutputs', 'all', 'Sensitivity');
```

```
f =  
SimFunction
```

Parameters:

Name	Value	Type
'Reaction1.c1'	10	'parameter'
'Reaction2.c2'	0.01	'parameter'

Observables:

Name	Type
'y1'	'species'
'y2'	'species'

Dosed: None

Sensitivity Input Factors:

Name	Type
'Reaction1.c1'	'parameter'
'Reaction2.c2'	'parameter'

Sensitivity Output Factors:

Name	Type
'y1'	'species'

```
'y2'    'species'
```

Sensitivity Normalization:

Full

Calculate sensitivities by executing the object with `c1` and `c2` set to 10 and 0.1 respectively. Set the output times from 1 to 10. `t` contains time points, `y` contains simulation data, and `sensMatrix` is the sensitivity matrix containing sensitivities of `y1` and `y2` with respect to `c1` and `c2`.

```
[t,y,sensMatrix] = f([10,0.1],[],[],1:10);
```

Retrieve the sensitivity information at simulation time = 5.

```
temp = sensMatrix{:};  
sensMatrix2 = temp(t{:}==5,,:);  
sensMatrix2 = squeeze(sensMatrix2)
```

```
sensMatrix2 = 2x2
```

```
    37.6987    -6.8447  
   -40.2791     5.8225
```

The rows of `sensMatrix2` represent output factors (`y1` and `y2`). The columns represent the input factors (`c1` and `c2`).

$$\mathit{sensMatrix2} = \begin{bmatrix} \frac{\partial y1}{\partial c1} & \frac{\partial y1}{\partial c2} \\ \frac{\partial y2}{\partial c1} & \frac{\partial y2}{\partial c2} \end{bmatrix}$$

Set the stop time to 15, without specifying the output times. In this case, the output times are the solver time points by default.

```
sd = f([10,0.1],15);
```

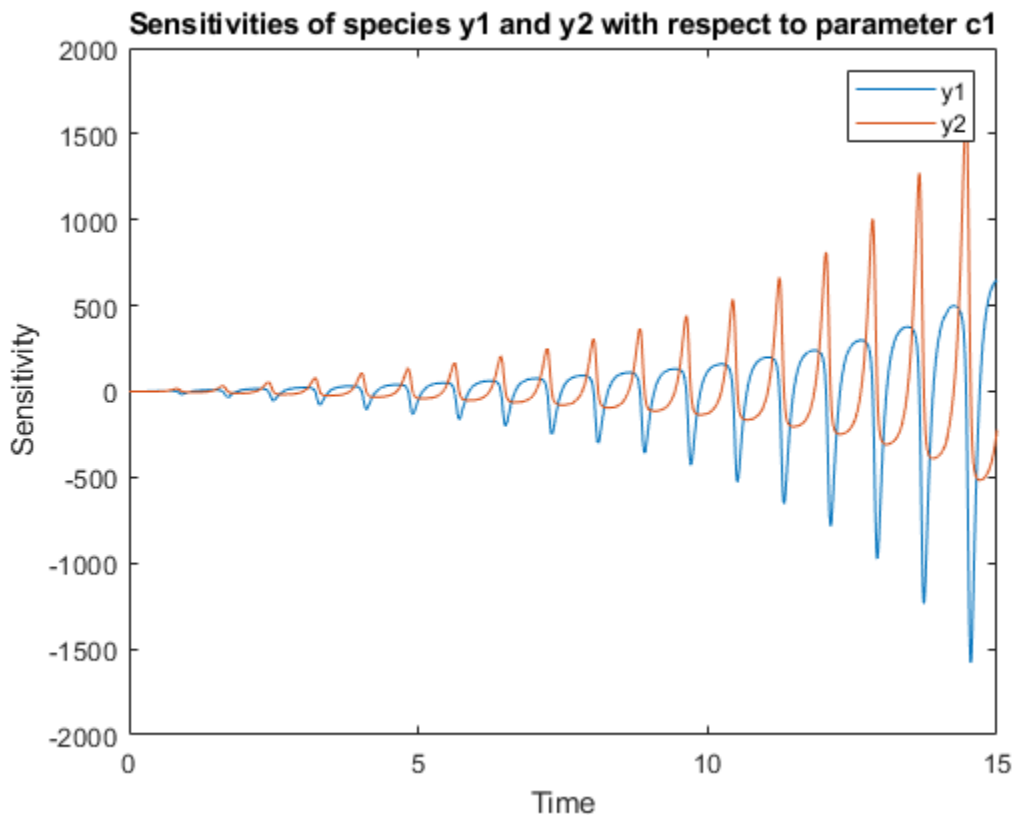
Retrieve the calculated sensitivities from the `SimData` object `sd`.

```
[t,y,outputs,inputs] = getsensmatrix(sd);
```



Plot the sensitivities of species y1 and y2 with respect to c1.

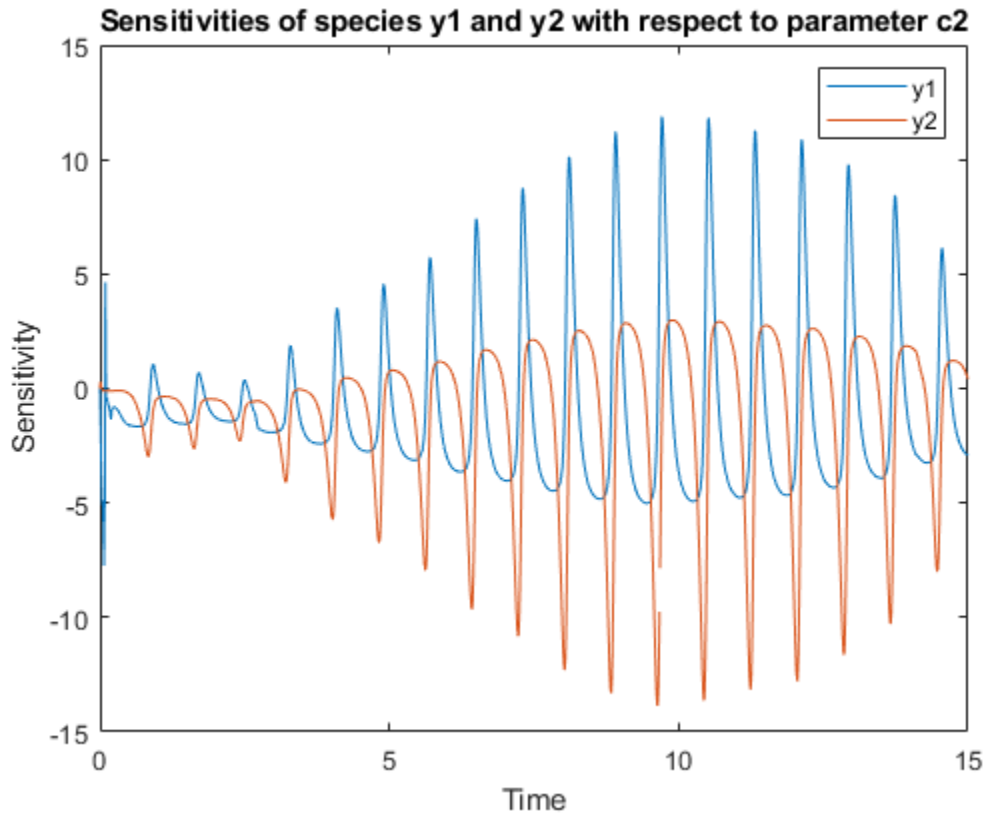
```
figure;  
plot(t,y(:,:,1));  
legend(outputs);  
title('Sensitivities of species y1 and y2 with respect to parameter c1');  
xlabel('Time');  
ylabel('Sensitivity');
```



Plot the sensitivities of species y1 and y2 with respect to c2.

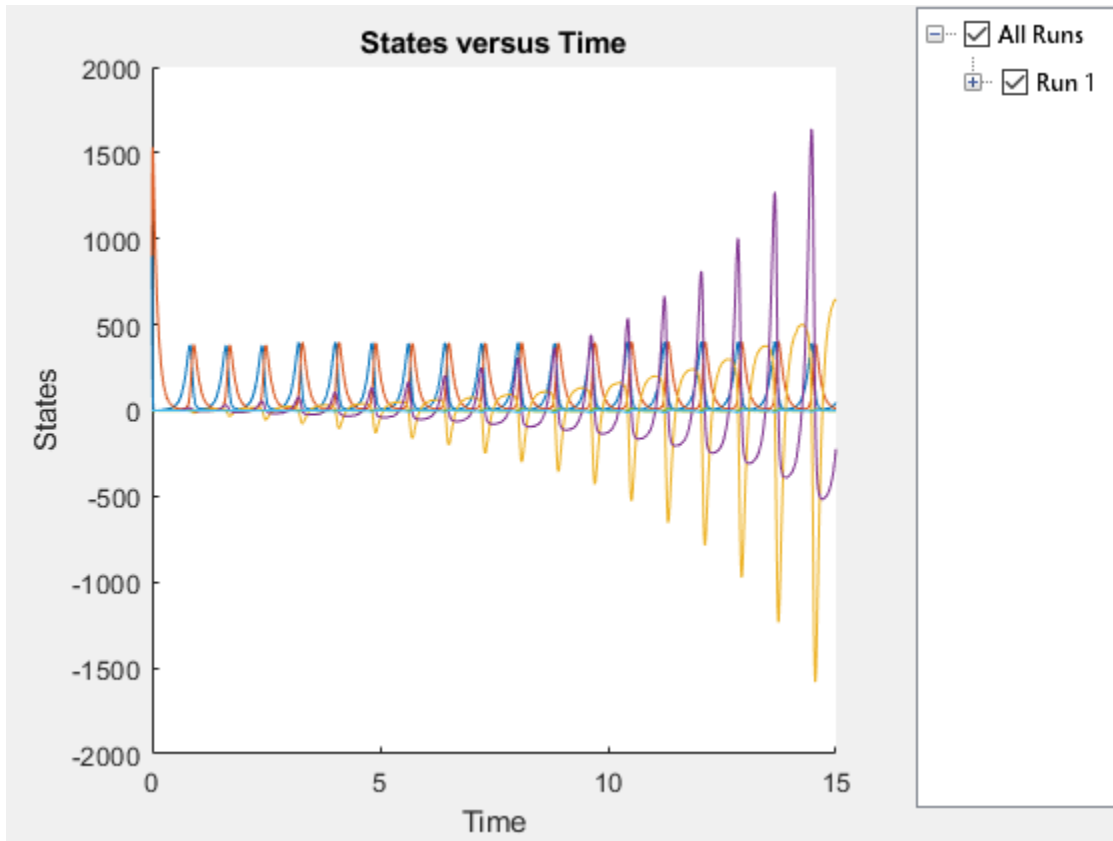
```
figure;  
plot(t,y(:,:,2));  
legend(outputs);
```

```
title('Sensitivities of species y1 and y2 with respect to parameter c2');  
xlabel('Time');  
ylabel('Sensitivity');
```



Alternatively, you can use `sbioplot`. Expand `Run1` to select which simulation or sensitivity data to display.

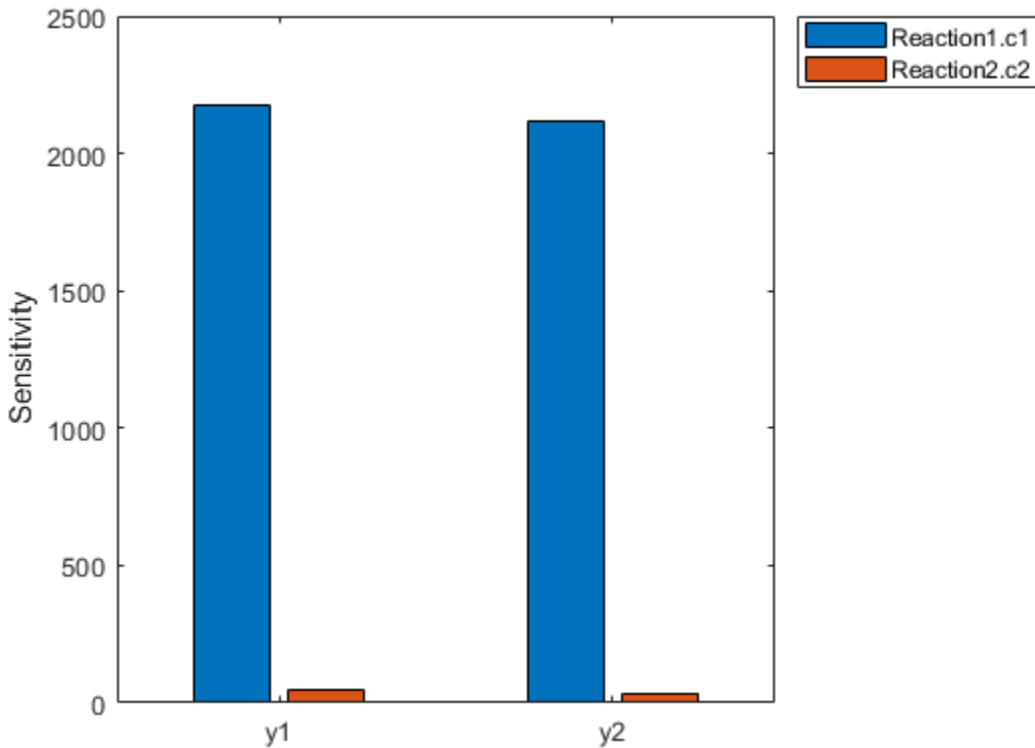
```
sbioplot(sd);
```



You can also plot the sensitivity matrix using the time integral for the calculated sensitivities of  $y_1$  and  $y_2$ . The plot indicates  $y_1$  and  $y_2$  are more sensitive to the parameter  $c_1$  than  $c_2$ .

```
[~, in, out] = size(y);
result = zeros(in, out);
for i = 1:in
    for j = 1:out
        result(i,j) = trapz(t(:),abs(y(:,i,j)));
    end
end
figure;
hbar = bar(result);
haxes = hbar(1).Parent;
```

```
haxes.XTick = 1:length(outputs);  
haxes.XTickLabel = outputs;  
legend(inputs, 'Location', 'NorthEastOutside');  
ylabel('Sensitivity');
```



### Simulate Model of Glucose-Insulin Response with Different Initial Conditions

This example shows how to simulate the glucose-insulin responses for the normal and diabetic subjects.

Load the model of glucose-insulin response. For details about the model, see the **Background** section in “Simulating the Glucose-Insulin Response”.

```
sbioloadproject('insulindemo', 'm1')
```

The model contains different initial conditions stored in various variants.

```
variants = getvariant(m1);
```

Get the initial conditions for the type 2 diabetic patient.

```
type2 = variants(1)
```

```
SimBiology Variant - Type 2 diabetic (inactive)
```

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	Plasma Volume ...	Value	1.49
2	parameter	k1	Value	0.042
3	parameter	k2	Value	0.071
4	parameter	Plasma Volume ...	Value	0.04
5	parameter	m1	Value	0.379
6	parameter	m2	Value	0.673
7	parameter	m4	Value	0.269
8	parameter	m5	Value	0.0526
9	parameter	m6	Value	0.8118
10	parameter	Hepatic Extrac...	Value	0.6
11	parameter	kmax	Value	0.0465
12	parameter	kmin	Value	0.0076
13	parameter	kabs	Value	0.023
14	parameter	kgri	Value	0.0465
15	parameter	f	Value	0.9
16	parameter	a	Value	6e-05
17	parameter	b	Value	0.68
18	parameter	c	Value	0.00023
19	parameter	d	Value	0.09
20	parameter	Stomach Glu Af...	Value	125
21	parameter	kp1	Value	3.09
22	parameter	kp2	Value	0.0007
23	parameter	kp3	Value	0.005
24	parameter	kp4	Value	0.0786
25	parameter	ki	Value	0.0066
26	parameter	[Ins Ind Glu U...	Value	1
27	parameter	Vm0	Value	4.65
28	parameter	Vmx	Value	0.034
29	parameter	Km	Value	466.21

30	parameter	p2U	Value	0.084
31	parameter	K	Value	0.99
32	parameter	alpha	Value	0.013
33	parameter	beta	Value	0.05
34	parameter	gamma	Value	0.5
35	parameter	ke1	Value	0.0007
36	parameter	ke2	Value	269
37	parameter	Basal Plasma G...	Value	164.18
38	parameter	Basal Plasma I...	Value	54.81

Suppress an informational warning that is issued during simulations.

```
warnSettings = warning('off', 'SimBiology:DimAnalysisNotDone_MatlabFcn_Dimensionless');
```

Create SimFunction objects to simulate the glucose-insulin response for the normal and diabetic subjects.

- Specify an empty array {} for the second input argument to denote that the model will be simulated using the base parameter values (that is, no parameter scanning will be performed).
- Specify the plasma glucose and insulin concentrations as responses (outputs of the function to be plotted).
- Specify the species Dose as the dosed species. This species represents the initial concentration of glucose at the start of the simulation.

```
normSim = createSimFunction(m1, {}, ...
    {'[Plasma Glu Conc]', '[Plasma Ins Conc]'}, 'Dose')
```

```
normSim =
SimFunction
```

Parameters:

Observables:

Name	Type	Units
'[Plasma Glu Conc]'	'species'	'milligram/deciliter'
'[Plasma Ins Conc]'	'species'	'picomole/liter'

Dosed:

TargetName	TargetDimension
------------	-----------------

Name	TargetDimension
'Dose'	'Mass (e.g., gram)'

For the diabetic patient, specify the initial conditions using the variant type2.

```
diabSim = createSimFunction(m1, {}, ...
    {'[Plasma Glu Conc]', '[Plasma Ins Conc]'}, 'Dose', type2)
```

```
diabSim =
SimFunction
```

Parameters:

Observables:

Name	Type	Units
'[Plasma Glu Conc]'	'species'	'milligram/deciliter'
'[Plasma Ins Conc]'	'species'	'picomole/liter'

Dosed:

TargetName	TargetDimension
'Dose'	'Mass (e.g., gram)'

Select a dose that represents a single meal of 78 grams of glucose at the start of the simulation.

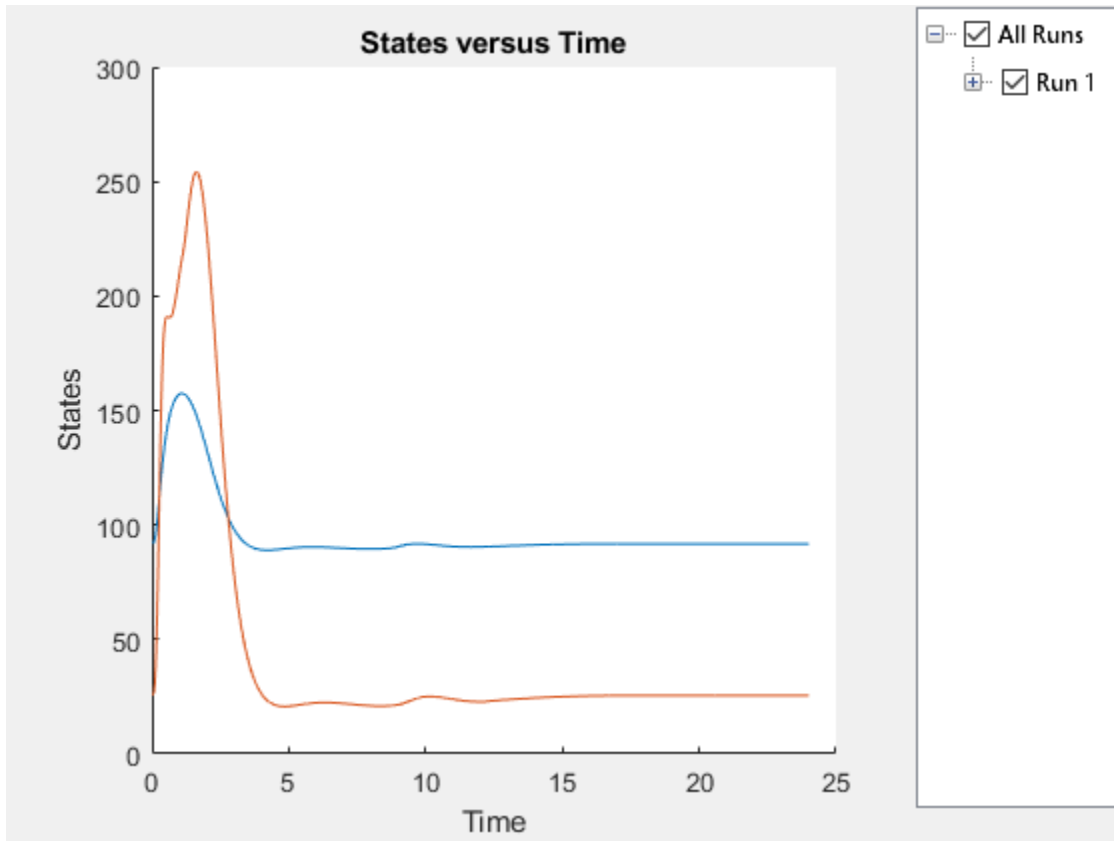
```
singleMeal = sbioselect(m1, 'Name', 'Single Meal');
```

Convert the dosing information to the table format.

```
mealTable = getTable(singleMeal);
```

Simulate the glucose-insulin response for a normal subject for 24 hours.

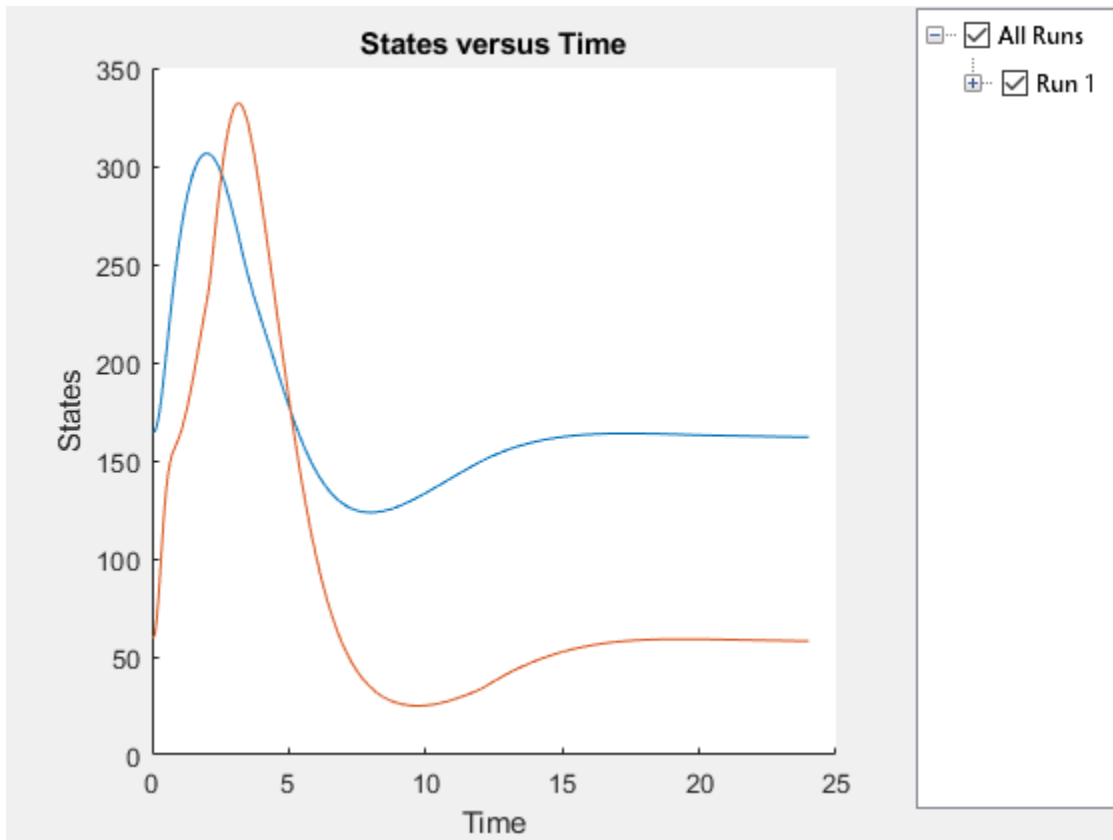
```
sbioplot(normSim([], 24, mealTable));
```



Simulate the glucose-insulin response for a diabetic subject for 24 hours.

```
sbioplot(diabSim([],24,mealTable));
```





### Perform a scan using variants

Suppose you want to perform a parameter scan using an array of variants that contain different initial conditions for different insulin impairments. For example, the model `m1` has variants that correspond to the low insulin sensitivity and high insulin sensitivity. You can simulate the model for both conditions via a single call to the `SimFunction` object.

Select the variants to scan.

```
varToScan = sbioselect(m1, 'Name', ...
    {'Low insulin sensitivity', 'High insulin sensitivity'});
```

Check which model parameters are being stored in each variant.

```
varToScan(1)
```

SimBiology Variant - Low insulin sensitivity (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	Vmx	Value	0.0235
2	parameter	kp3	Value	0.0045

varToScan(2)

SimBiology Variant - High insulin sensitivity (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	Vmx	Value	0.094
2	parameter	kp3	Value	0.018

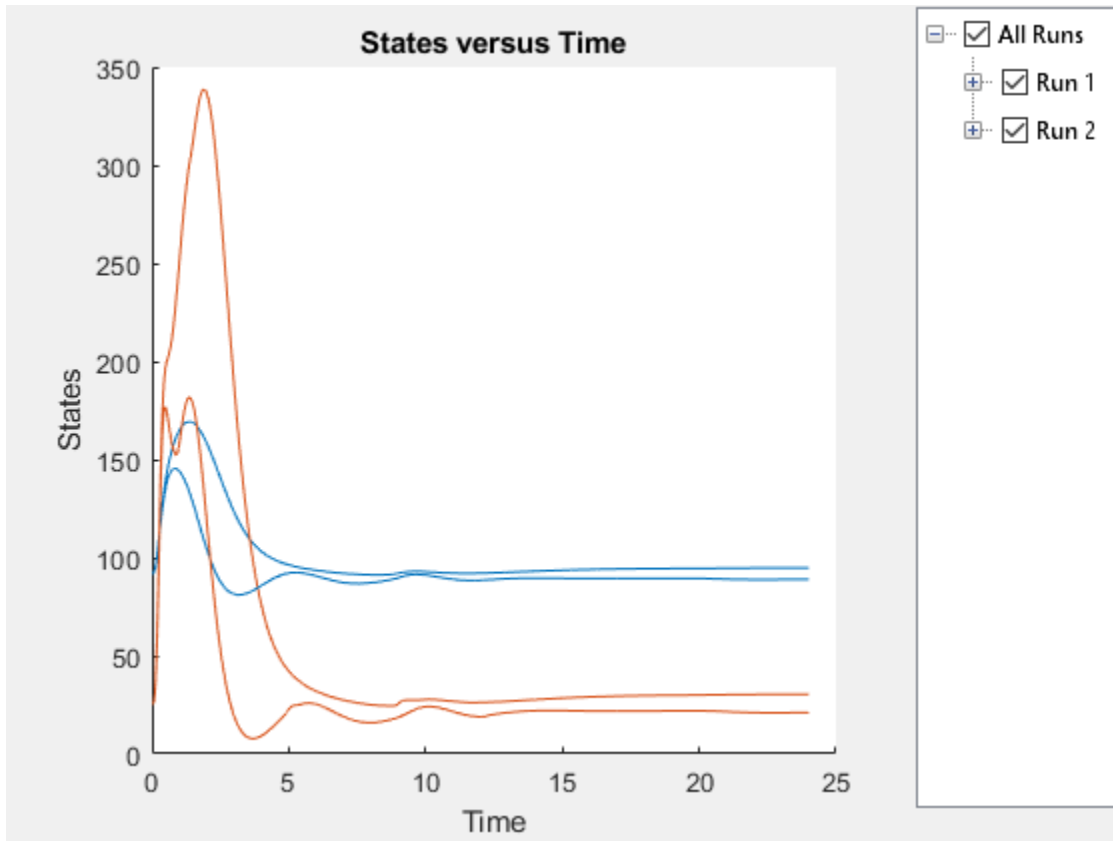
Both variants store alternate values for Vmx and kp3 parameters. You need to specify them as input parameters when you create a SimFunction object.

Create a SimFunction object to scan the variants.

```
variantScan = createSimFunction(m1, {'Vmx', 'kp3'}, ...  
    {'[Plasma Glu Conc]', '[Plasma Ins Conc]', 'Dose'});
```

Simulate the model and plot the results. Run 1 include simulation results for the low insulin sensitivity and Run 2 for the high insulin sensitivity.

```
sbioplot(variantScan(varToScan, 24, mealTable));
```



Low insulin sensitivity lead to increased and prolonged plasma glucose concentration.

Restore warning settings.

```
warning(warnSettings);
```

## References

[1] Gillespie, D.T. (1977). Exact Stochastic Simulation of Coupled Chemical Reactions. The Journal of Physical Chemistry. 81(25), 2340-2361.

## Extended Capabilities

### Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set 'UseParallel' to true.

For more information, see the 'UseParallel' name-value pair argument.

### See Also

`SimFunction` object | `SimFunctionSensitivity` object | `model` object | `sbiosampleerror` | `sbiosampleparameters`

### Topics

“Model Simulation”

**Introduced in R2014a**

## delete (any object)

Delete SimBiology object

### Syntax

```
delete(Obj)
```

### Arguments

<i>Obj</i>	abstract kinetic law, configuration set, event, kinetic law, model, parameter, reaction, rule, SimData, species, unit, unit prefix, or variant object.
------------	--

### Description

`delete(Obj)` removes an object (*Obj*) from its parent.

- If *Obj* is a model object, the model is deleted from the root object. `delete` removes all references to the model both at the command line and in the SimBiology desktop.
- If *Obj* is a species object that is being used by a reaction object, this method returns an error and the species object is not deleted. You need to delete the reaction or remove the species from the reaction before you can delete the species object.
- If *Obj* is a parameter object being used by a kinetic law object, there is no warning when the object is deleted. However, when you try to simulate your model, a error occurs because the parameter cannot be found.
- If *Obj* is a reaction object, this method deletes the object, but the species objects that were being used by the reaction object are not deleted.
- If *Obj* is an abstract kinetic law object and there is a kinetic law object referencing it, this method returns an error.

- If *Obj* is a SimBiology configuration set object, and it is the active configuration set object, this method, after deleting the object, makes the default configuration set object active. Note that you cannot delete the default configuration set.
- You cannot delete the SimBiology root.

You can also delete all model objects from the root with one call to the `sbioreset` function.

## Examples

### Example 1

Delete a reaction from a model. Notice the species objects are not deleted with the reaction object.

```
modelObj = sbiomodel('cell');  
reactionObj = addreaction(modelObj, 'a -> b');  
delete(reactionObj)
```

### Example 2

Delete a single model from the root object.

```
modelObj1 = sbiomodel('cell');  
modelObj2 = sbiomodel('virus');  
delete(modelObj2)
```

## See Also

`sbiomodel`, `sbioreset`, `sbioroot`

**Introduced in R2006a**

## display (any object)

Display summary of SimBiology object

### Syntax

```
display(Obj)
```

### Arguments

<i>Obj</i>	SimBiology object: abstract kinetic law, configuration set, compartment, event, kinetic law, model, parameter, reaction, rule, species, or unit.
------------	--

### Description

Display the SimBiology object array. `display(Obj)` is called for the SimBiology object, *Obj* when the semicolon is not used to terminate a statement. The display of *Obj* gives a brief summary of the *Obj* configuration. You can view a complete list of *Obj* properties with the command `get`. You can modify all *Obj* properties that can be changed, with the command `set`.

### Examples

```
modelObj = sbiomodel('cell')  
reactionObj = addreaction(modelObj, 'A + B -> C')
```

**Introduced in R2006a**

## EstimatedInfo object

Object containing information about estimated model quantities

### Description

The `estimatedInfo` object contains information about estimated model quantities (species, parameters, or compartments). Use this object to specify which quantities in a SimBiology model are estimated, what parameter transforms are used, and optionally, the initial estimates for these quantities. Use this object to specify what quantities in a SimBiology model are estimated when using `sbiofit` or `sbiofitmixed`.

### Construction

`estimInfo = estimatedInfo` creates an empty `estimatedInfo` object.

`estimInfoArray = estimatedInfo(transformedNames)` creates an array of `estimatedInfo` objects for quantities specified in `transformedNames`. The initial values for these quantities are obtained from the SimBiology model when you run `sbiofit` or `sbiofitmixed`.

`estimInfoArray = estimatedInfo( ____, 'InitialTransformedValue', itValues)` defines the initial transformed values of model quantities specified by `itValues`. You cannot specify this name-value pair along with the 'InitialValue' name-value pair.

`estimInfoArray = estimatedInfo( ____, 'InitialValue', iValues)` defines the initial values of model quantities specified by `iValues`. You cannot specify this name-value pair along with the 'InitialTransformedValue' name-value pair.

`estimInfoArray = estimatedInfo( ____, 'Bounds', boundValues)` defines the lower and upper bounds for parameter estimation specified by `boundValues`. You cannot specify this name-value pair along with the 'TransformedBounds' name-value pair. All methods support parameter bounds in `sbiofit` (that is, `fminsearch`, `nlinfit`, `fminunc`, `fmincon`, `lsqcurvefit`, `lsqnonlin`, `patternsearch`, `ga`, `particleswarm`, and `scattersearch` on page 1-109). When using `fminsearch`, `nlinfit`, or `fminunc` with bounds, the objective function returns `Inf` if bounds are exceeded. When you turn on



options such as `FunValCheck`, the optimization may error if bounds are exceeded during estimation. If using `nlinfit`, it may report warnings about the Jacobian being ill-conditioned or not being able to estimate if the final result is too close to the bounds. `sbiofitmixed` does not support parameter bounds.

`estimInfoArray = estimatedInfo( ____, 'TransformedBounds', tBoundValues)`  
 defines the transformed bounds for parameter estimation specified by `tBoundValues`. You cannot specify this name-value pair along with the 'Bounds' name-value pair.

`estimInfoArray = estimatedInfo( ____, 'CategoryVariableName', groups)`  
 defines which groups in the data have separate estimated values for parameters. In other words, this allows you to estimate parameter values specific for each group or category. For example, you can estimate parameters based on individuals' age or sex.

## Input Arguments

### **transformedNames** — Names of estimated model quantities

character vector | string | string vector | cell array of character vectors

Names of estimated model quantities, specified as a character vector, string, string vector, or cell array of character vectors. Each character vector (or string) must be in one of these formats:

- Name or qualified name of a model quantity, such as 'Cl', 'Reaction1.k', '[c 1].[r 1]'
- Name of a supported parameter transform (`log`, `logit`, or `probit`) followed by a quantity name in parentheses, such as 'log(Cl)', 'logit(Reaction1.k)', 'probit([c 1].[r 1])'

### **itValues** — Initial transformed values of model quantities

vector | cell array

Initial transformed values of model quantities, specified as a vector or cell array. It must have the same length as `transformedNames`. If it is a cell array, each element of the cell must be a scalar or the empty matrix `[]`.

### **iValues** — Initial values of model quantities

vector | cell array

Initial values of model quantities, specified as a vector or cell array. It must have the same length as `transformedNames`. If it is a cell array, each element of the cell must be a scalar or the empty matrix `[]`.

### **boundValues — Bound constraints for estimated parameters**

`[]` (default) | matrix | cell array

Bound constraints for estimated parameters, specified as a matrix or cell array. If `boundValues` is a matrix, it is a  $N$ -by-2 matrix of numbers, where  $N$  is either 1 or the length of `transformedNames`. If it is a cell array, each element must be a vector of size 1-by-2 or empty `[]`.

Each row of `boundValues` corresponds to the lower (the first number) and upper (the second number) bounds of each element (such as a parameter) of `estimInfo`. The lower bound must be less than the upper bound. If you specify a single row, these bounds are applied to all elements of `estimInfoArray`.

You cannot specify this name-value pair along with the 'TransformedBounds' name-value pair.

### **tBoundValues — Transformed bound constraints for estimated parameters**

`[]` (default) | matrix | cell array

Transformed bound constraints for estimated parameters, specified as a matrix or cell array. `tBoundValues` is a  $N$ -by-2 matrix of numbers, where  $N$  is either 1 or the length of `transformedNames`. If it is a cell array, each element must be a vector of size 1-by-2 or empty `[]`.

Each row of `tBoundValues` corresponds to the lower (the first number) and upper (the second number) bounds of each element (such as a parameter) of `estimInfo`. The lower bound must be less than the upper bound. If you specify a single row, the bounds are applied to all elements of `estimInfoArray`.

You cannot specify this name-value pair along with the 'Bounds' name-value pair.

### **groups — Group names for estimated parameters**

character vector | string | string vector | cell array of character vectors

Group names for estimated parameters, specified as a character vector, string, string vector, or cell array of character vectors. Each character vector (or string) must be one of the following.

- Name of a variable in the data used for fitting
- '<GroupVariableName>' (default)
- '<None>'

'<GroupVariableName>' indicates that each group in the data uses a separate parameter estimate. '<None>' indicates that all groups in the data use the same parameter estimate.

If the data you plan to use for fitting contains variables that group data into different categories, you can specify the names of those variables. For instance, if you have a variable called `Sex` which indicates male and female individuals, you can specify `'Sex'` as the `'CategoryVariableName'`. This means that all male individuals have one set of parameter estimates and all females have a separate set.

## Output Arguments

### **estimInfo** — Estimated model quantity

`estimatedInfo` object

Estimated model quantity, returned as an `estimatedInfo` object.

### **estimInfoArray** — Estimated model quantities

`estimatedInfo` object | vector

Estimated model quantities, returned as an `estimatedInfo` object or vector of `estimatedInfo` objects. If `transformedNames` is a single character vector, `estimInfoArray` is a scalar `estimatedInfo` object. Otherwise, `estimInfoArray` is a vector of `estimatedInfo` objects with the same length as the input argument `transformedNames`.

## Property Summary

Name	Character vector indicating the name of an estimated model quantity. Changing this property also updates the <code>TransformedName</code> property.
------	---

<p><b>Transform</b></p>	<p>Character vector indicating whether the quantity value is transformed during estimation. Valid names are '', 'log', 'logit', and 'probit'. An empty character vector '' indicates that no transform is applied.</p> <p>A log transform ensures that the component value is always positive during estimation. The <code>logit</code> and <code>probit</code> transforms constrain component values to lie between 0 and 1.</p> <p>The <code>probit</code> function is the inverse cumulative distribution function associated with the standard normal distribution. For the <code>probit</code> transform, SimBiology uses the <code>norminv</code> function. Hence Statistics and Machine Learning Toolbox is required for the transform.</p> <p>The logit function, which is the inverse of sigmoid function, is defined as <math>\text{logit}(x) = \log(x) - \log(1 - x)</math>.</p>
<p><b>TransformedName</b></p>	<p>Read-only character vector that combines the transform name (such as 'log') and the quantity name (such as 'Central') into an expression (such as 'log(Central)').</p>
<p><b>InitialValue</b></p>	<p>Empty matrix [] or real, finite, scalar value specifying the initial values of model quantities used for estimation. The empty matrix indicates that the initial values for estimation are obtained from the relevant quantity property (<code>Value</code> for parameters, <code>InitialAmount</code> for species, and <code>Capacity</code> for compartments).</p> <p>Changing this property automatically updates the <code>InitialTransformedValue</code> property of corresponding model quantities.</p>

InitialTransformedValue	<p>Empty matrix [] or scalar value specifying the initial transformed values of model quantities used for estimation. The empty matrix indicates that the initial transformed values for estimation are obtained by transforming the relevant quantity property (Value for parameters, InitialAmount for species, and Capacity for compartments).</p> <p>Changing this property automatically updates the InitialValue property of corresponding model quantities.</p>
Bounds	<p>Empty matrix [] or 1-by-2 vector of real, finite value specifying the lower and upper bound for an estimated parameter. The empty matrix [] indicates that the only bound constraints are those introduced by the value of Transform. For example, setting Transform to 'log' constrains the parameter to the range [0, inf].</p> <p>Changing this property also updates TransformedBounds.</p> <p>The lower bound must be less than the upper bound.</p>
TransformedBounds	<p>Empty matrix [] or 1-by-2 vector of real, finite value specifying the lower and upper bound for an estimated parameter. The empty matrix [] indicates that the value of the parameter in transformed space is not constrained.</p> <p>Changing this property also updates Bounds.</p> <p>The lower bound must be less than the upper bound.</p>

CategoryVariableName	<p>Character vector or cell array of character vectors specifying which groups in the data have separate estimated values for the parameter. The character vector can be the name of a variable in the data used for fitting or one of the keywords: '&lt;GroupVariableName&gt;' or '&lt;None&gt;'.</p> <p>'&lt;GroupVariableName&gt;' indicates that each group in the data uses a separate parameter estimate. '&lt;None&gt;' indicates that all groups in the data use the same parameter estimate.</p> <p>If you specify 'Pooled' name-value pair argument (to either true or false) when you run <code>sbiofit</code>, then the function ignores this variable. <code>sbiofitmixed</code> always ignores this property.</p>
----------------------	--

## Examples

### Specify Estimated Parameters Using an EstimatedInfo Object

Create a one-compartment PK model with bolus dosing and linear clearance.

```
pkmd           = PKModelDesign;
pkc1          = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
```

Suppose you want to estimate the volume of the central compartment (`Central`). You can specify such estimated model quantity as well as appropriate parameter transform (log transform in this example), initial value, and parameter bound using the `estimatedInfo` object.

```
estimated = estimatedInfo('log(Central)', 'InitialValue', 1, 'Bounds', [0 10])
```

```
estimated =
    estimatedInfo with properties:
```

```
        Name: 'Central'
```

```

        Transform: 'log'
    TransformedName: 'log(Central)'
        InitialValue: 1
    InitialTransformedValue: 0
            Bounds: [0 10]
    TransformedBounds: [-Inf 2.3026]
    CategoryVariableName: '<GroupVariableName>'

```

## Fit a One-Compartment Model to an Individual's PK Profile

### Background

This example shows how to fit an individual's PK profile data to one-compartment model and estimate pharmacokinetic parameters.

Suppose you have drug plasma concentration data from an individual and want to estimate the volume of the central compartment and the clearance. Assume the drug

concentration versus the time profile follows the monoexponential decline  $C_t = C_0 e^{-k_e t}$ , where  $C_t$  is the drug concentration at time  $t$ ,  $C_0$  is the initial concentration, and  $k_e$  is the elimination rate constant that depends on the clearance and volume of the central compartment  $k_e = Cl/V$ .

The synthetic data in this example was generated using the following model and parameters:

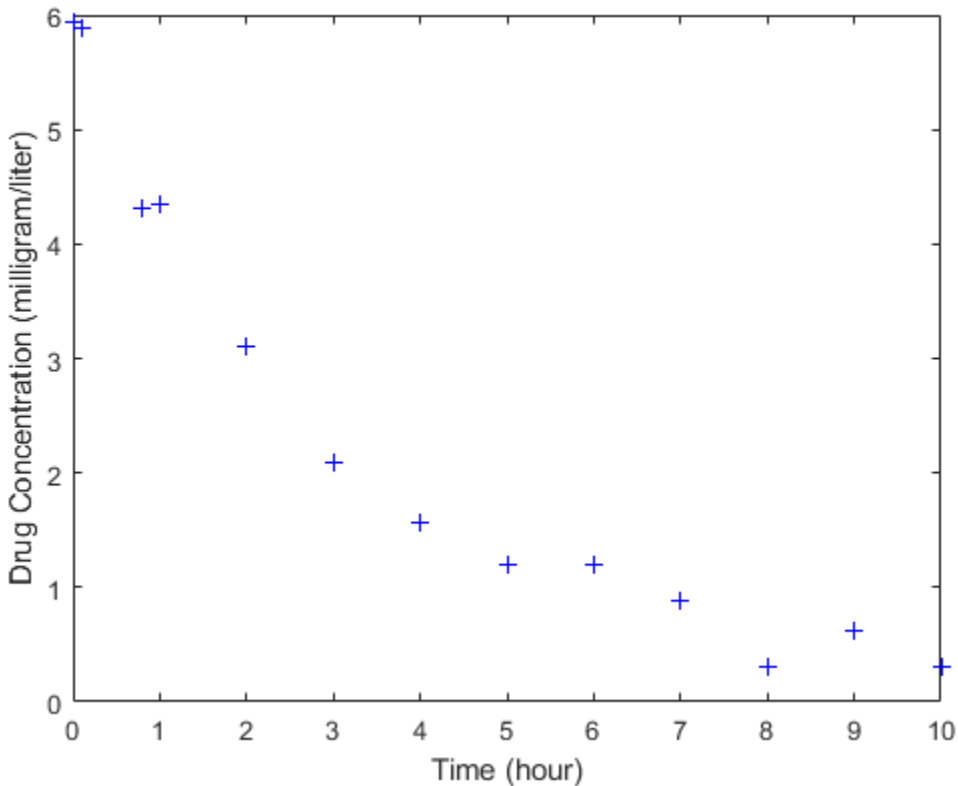
- One-compartment model with bolus dosing and first-order elimination
- Volume of the central compartment (Central) = 1.70 liter
- Clearance parameter (Cl\_Central) = 0.55 liter/hour
- Constant error model

### Load Data and Visualize

The data is stored as a table with variables Time and Conc that represent the time course of the plasma concentration of an individual after an intravenous bolus administration measured at 13 different time points. The variable units for Time and Conc are hour and milligram/liter, respectively.

```
clear all
load(fullfile(matlabroot,'examples','simbio','data15.mat'))

plot(data.Time,data.Conc,'b+')
xlabel('Time (hour)');
ylabel('Drug Concentration (milligram/liter)');
```



### Convert to groupedData Format

Convert the data set to a `groupedData` object, which is the required data format for the fitting function `sbiofit` for later use. A `groupedData` object also lets you set independent variable and group variable names (if they exist). Set the units of the `Time` and `Conc` variables. The units are optional and only required for the `UnitConversion`



feature, which automatically converts matching physical quantities to one consistent unit system.

```
gData = groupedData(data);
gData.Properties.VariableUnits = {'hour', 'milligram/liter'};
gData.Properties

ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Row' 'Variables'}
    VariableNames: {'Time' 'Conc'}
    VariableDescriptions: {}
    VariableUnits: {'hour' 'milligram/liter'}
    VariableContinuity: []
    RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: ''
    IndependentVariableName: 'Time'
```

`groupedData` automatically set the name of the `IndependentVariableName` property to the `Time` variable of the data.

### Construct a One-Compartment Model

Use the built-in PK library to construct a one-compartment model with bolus dosing and first-order elimination where the elimination rate depends on the clearance and volume of the central compartment. Use the `configset` object to turn on unit conversion.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
model = construct(pkmd);
configset = getconfigset(model);
configset.CompileOptions.UnitConversion = true;
```

For details on creating compartmental PK models using the SimBiology® built-in library, see “Create Pharmacokinetic Models”.

### Define Dosing

Define a single bolus dose of 10 milligram given at time = 0. For details on setting up different dosing schedules, see “Doses”.

```
dose                = sbiodose('dose');
dose.TargetName     = 'Drug_Central';
dose.StartTime      = 0;
dose.Amount         = 10;
dose.AmountUnits    = 'milligram';
dose.TimeUnits      = 'hour';
```

### Map Response Data to the Corresponding Model Component

The data contains drug concentration data stored in the `Conc` variable. This data corresponds to the `Drug_Central` species in the model. Therefore, map the data to `Drug_Central` as follows.

```
responseMap = {'Drug_Central = Conc'};
```

### Specify Parameters to Estimate

The parameters to fit in this model are the volume of the central compartment (`Central`) and the clearance rate (`Cl_Central`). In this case, specify log-transformation for these biological parameters since they are constrained to be positive. The `estimatedInfo` object lets you specify parameter transforms, initial values, and parameter bounds if needed.

```
paramsToEstimate    = {'log(Central)', 'log(Cl_Central)'};
estimatedParams     = estimatedInfo(paramsToEstimate, 'InitialValue', [1 1], 'Bounds', [1 5]);
```

### Estimate Parameters

Now that you have defined one-compartment model, data to fit, mapped response data, parameters to estimate, and dosing, use `sbiofit` to estimate parameters. The default estimation function that `sbiofit` uses will change depending on which toolboxes are available. To see which function was used during fitting, check the `EstimationFunction` property of the corresponding results object.

```
fitConst = sbiofit(model, gData, responseMap, estimatedParams, dose);
```

## Display Estimated Parameters and Plot Results

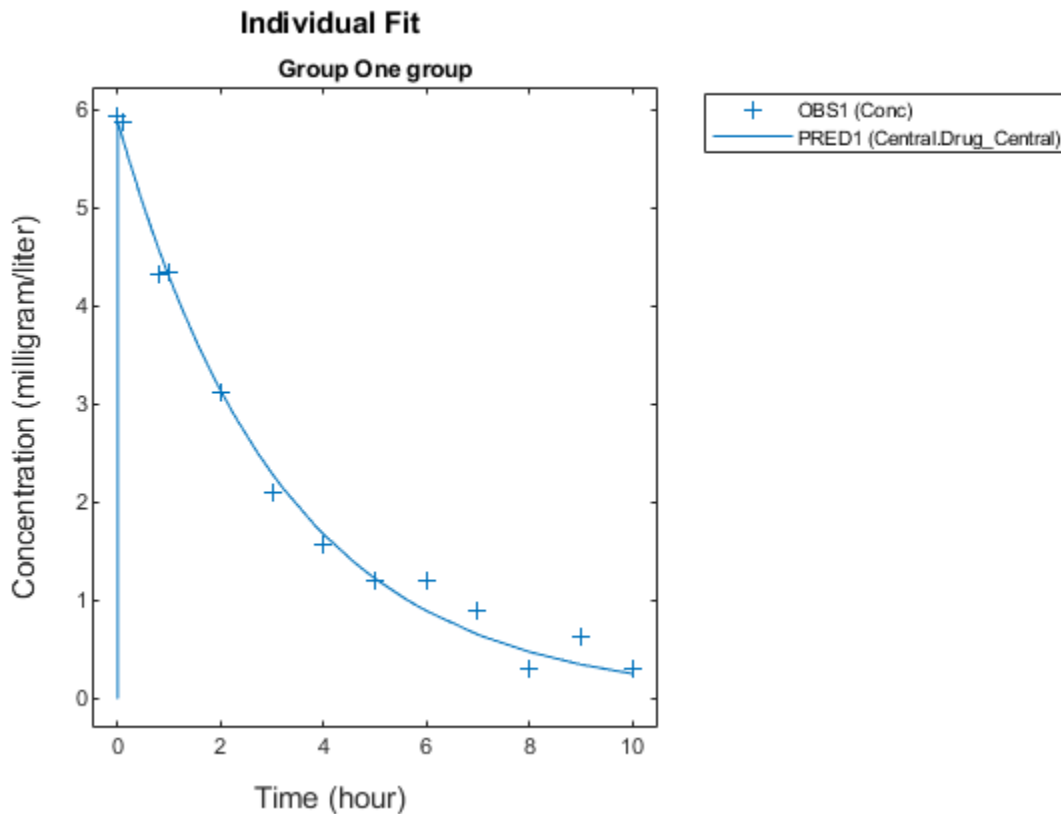
Notice the parameter estimates were not far off from the true values (1.70 and 0.55) that were used to generate the data. You may also try different error models to see if they could further improve the parameter estimates.

```
fitConst.ParameterEstimates
```

```
ans=2x4 table
```

Name	Estimate	StandardError	Bounds	
'Central'	1.6993	0.034821	1	5
'Cl_Central'	0.53358	0.01968	0.5	2

```
s.Labels.XLabel = 'Time (hour)';  
s.Labels.YLabel = 'Concentration (milligram/liter)';  
plot(fitConst, 'AxesStyle', s);
```



### Use Different Error Models

Try three other supported error models (proportional, combination of constant and proportional error models, and exponential).

```
fitProp = sbiofit(model,gData,responseMap,estimatedParams,dose,...  
                'ErrorModel','proportional');  
fitExp  = sbiofit(model,gData,responseMap,estimatedParams,dose,...  
                'ErrorModel','exponential');  
fitComb = sbiofit(model,gData,responseMap,estimatedParams,dose,...  
                'ErrorModel','combined');
```

## Use Weights Instead of an Error Model

You can specify weights as a numeric matrix, where the number of columns corresponds to the number of responses. Setting all weights to 1 is equivalent to the constant error model.

```
weightsNumeric = ones(size(gData.Conc));
fitWeightsNumeric = sbiofit(model,gData,responseMap,estimatedParams,dose,'Weights',wei
```

Alternatively, you can use a function handle that accepts a vector of predicted response values and returns a vector of weights. In this example, use a function handle that is equivalent to the proportional error model.

```
weightsFunction = @(y) 1./y.^2;
fitWeightsFunction = sbiofit(model,gData,responseMap,estimatedParams,dose,'Weights',wei
```

## Compare Information Criteria for Model Selection

Compare the loglikelihood, AIC, and BIC values of each model to see which error model best fits the data. A larger likelihood value indicates the corresponding model fits the model better. For AIC and BIC, the smaller values are better.

```
allResults = [fitConst,fitWeightsNumeric,fitWeightsFunction,fitProp,fitExp,fitComb];
errorModelNames = {'constant error model','equal weights','proportional weights', ...
                   'proportional error model','exponential error model',...
                   'combined error model'};
LogLikelihood = [allResults.LogLikelihood]';
AIC = [allResults.AIC]';
BIC = [allResults.BIC]';
t = table(LogLikelihood,AIC,BIC);
t.Properties.RowNames = errorModelNames;
t
```

t=6×3 table

	LogLikelihood	AIC	BIC
constant error model	3.9866	-3.9732	-2.8433
equal weights	3.9866	-3.9732	-2.8433
proportional weights	-3.8472	11.694	12.824
proportional error model	-3.8257	11.651	12.781
exponential error model	1.1984	1.6032	2.7331
combined error model	3.9163	-3.8326	-2.7027

Based on the information criteria, the constant error model (or equal weights) fits the data best since it has the largest loglikelihood value and the smallest AIC and BIC.

### Display Estimated Parameter Values

Show the estimated parameter values of each model.

```
Estimated_Central      = zeros(6,1);
Estimated_Cl_Central  = zeros(6,1);
t2 = table(Estimated_Central,Estimated_Cl_Central);
t2.Properties.RowNames = errorModelNames;
for i = 1:height(t2)
    t2{i,1} = allResults(i).ParameterEstimates.Estimate(1);
    t2{i,2} = allResults(i).ParameterEstimates.Estimate(2);
end
```

`t2`

*t2=6x2 table*

	Estimated_Central	Estimated_Cl_Central
constant error model	1.6993	0.53358
equal weights	1.6993	0.53358
proportional weights	1.9045	0.51734
proportional error model	1.8777	0.51147
exponential error model	1.7872	0.51701
combined error model	1.7008	0.53271

### Conclusion

This example showed how to estimate PK parameters, namely the volume of the central compartment and clearance parameter of an individual, by fitting the PK profile data to one-compartment model. You compared the information criteria of each model and estimated parameter values of different error models to see which model best explained the data. Final fitted results suggested both the constant and combined error models provided the closest estimates to the parameter values used to generate the data. However, the constant error model is a better model as indicated by the loglikelihood, AIC, and BIC information criteria.

## Estimate Category-Specific PK Parameters for Multiple Individuals

This example shows how to estimate category-specific (such as young versus old, male versus female), individual-specific, and population-wide parameters using PK profile data from multiple individuals.

### Background

Suppose you have drug plasma concentration data from 30 individuals and want to estimate pharmacokinetic parameters, namely the volumes of central and peripheral compartment, the clearance, and intercompartmental clearance. Assume the drug concentration versus the time profile follows the biexponential decline

$C_t = Ae^{-at} + Be^{-bt}$ , where  $C_t$  is the drug concentration at time  $t$ , and  $a$  and  $b$  are slopes for corresponding exponential declines.

### Load Data

This synthetic data contains the time course of plasma concentrations of 30 individuals after a bolus dose (100 mg) measured at different times for both central and peripheral compartments. It also contains categorical variables, namely Sex and Age.

```
clear
load(fullfile(matlabroot, 'examples', 'simbio', 'sd5_302RAgeSex.mat'))
```

### Convert to groupedData Format

Convert the data set to a `groupedData` object, which is the required data format for the fitting function `sbiofit`. A `groupedData` object also allows you set independent variable and group variable names (if they exist). Set the units of the ID, Time, CentralConc, PeripheralConc, Age, and Sex variables. The units are optional and only required for the `UnitConversion` feature, which automatically converts matching physical quantities to one consistent unit system.

```
gData = groupedData(data);
gData.Properties.VariableUnits = {'', 'hour', 'milligram/liter', 'milligram/liter', '', ''};
gData.Properties
```

```
ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Row' 'Variables'}
    VariableNames: {1x6 cell}
    VariableDescriptions: {}
```

```
        VariableUnits: {1x6 cell}
VariableContinuity: []
        RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: 'ID'
IndependentVariableName: 'Time'
```

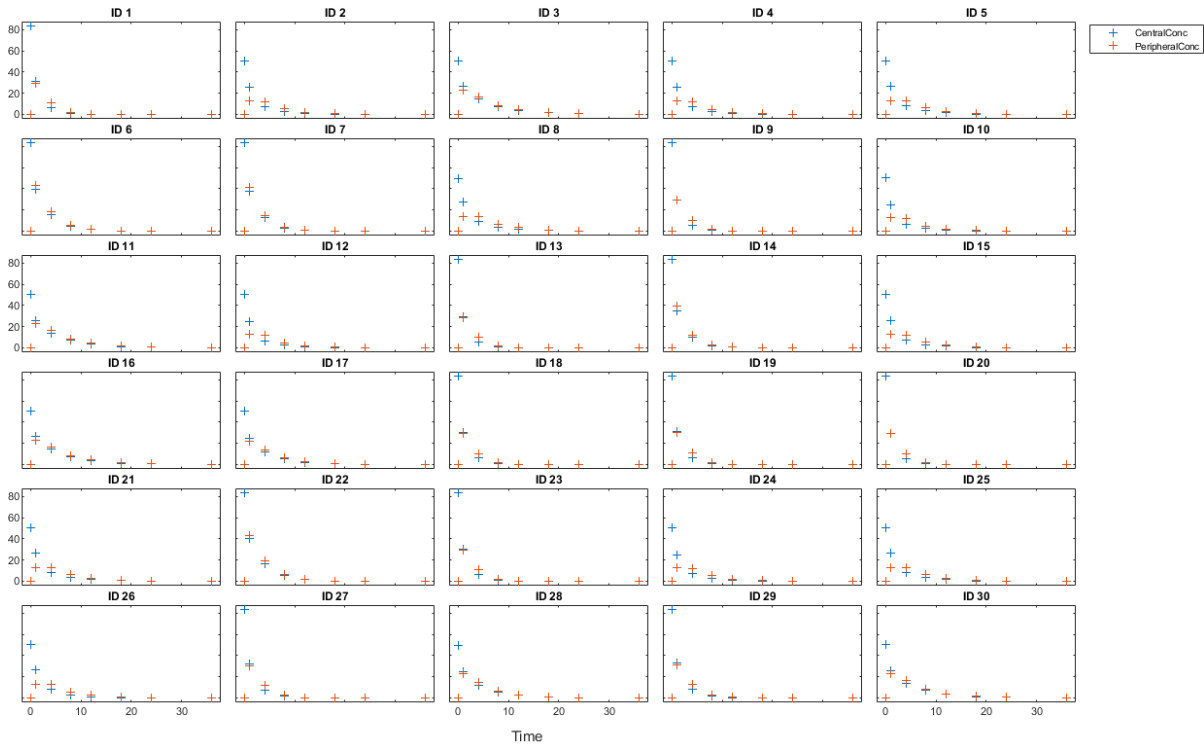
The `IndependentVariableName` and `GroupVariableName` properties have been automatically set to the `Time` and `ID` variables of the data.

### Visualize Data

Display the response data for each individual.

```
t = sbiotrellis(gData, 'ID', 'Time', {'CentralConc', 'PeripheralConc'}, ...
    'Marker', '+', 'LineStyle', 'none');
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```





## Set Up a Two-Compartment Model

Use the built-in PK library to construct a two-compartment model with infusion dosing and first-order elimination where the elimination rate depends on the clearance and volume of the central compartment. Use the `configset` object to turn on unit conversion.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc2 = addCompartment(pkmd, 'Peripheral');
model = construct(pkmd);
configset = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

For details on creating compartmental PK models using the SimBiology® built-in library, see “Create Pharmacokinetic Models”.

### Define Dosing

Assume every individual receives a bolus dose of 100 mg at time = 0. For details on setting up different dosing strategies, see “Doses”.

```
dose                = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime     = 0;
dose.Amount        = 100;
dose.AmountUnits   = 'milligram';
dose.TimeUnits     = 'hour';
```

### Map the Response Data to Corresponding Model Components

The data contains measured plasma concentration in the central and peripheral compartments. Map these variables to the appropriate model components, which are `Drug_Central` and `Drug_Peripheral`.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
```

### Specify Parameters to Estimate

Specify the volumes of central and peripheral compartments `Central` and `Peripheral`, intercompartmental clearance `Q12`, and clearance `Cl_Central` as parameters to estimate. The `estimatedInfo` object lets you optionally specify parameter transforms, initial values, and parameter bounds. Since both `Central` and `Peripheral` are constrained to be positive, specify a log-transform for each parameter.

```
paramsToEstimate    = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam      = estimatedInfo(paramsToEstimate, 'InitialValue', [1 1 1 1]);
```

### Estimate Individual-Specific Parameters

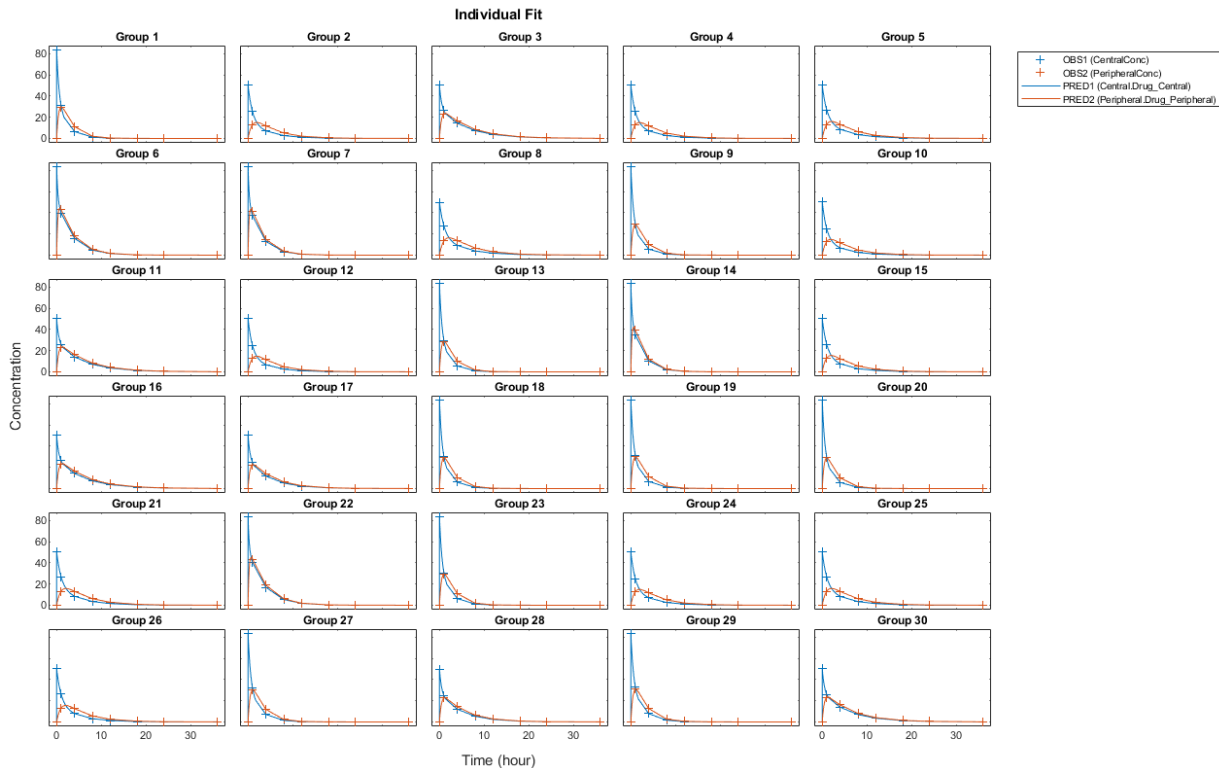
Estimate one set of parameters for each individual by setting the `'Pooled'` name-value pair argument to `false`.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

### Display Results

Plot the fitted results versus the original data for each individual (group).

```
t = plot(unpooledFit);
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```



For an unpooled fit, `sbiofit` always returns one results object for each individual.

### Examine Parameter Estimates for Category Dependencies

Explore the unpooled estimates to see if there is any category-specific parameters, that is, if some parameters are related to one or more categories. If there are any category dependencies, it might be possible to reduce the number of degrees of freedom by estimating just category-specific values for those parameters.

First extract the ID and category values for each ID

```
catParamValues = unique(gData(:, {'ID', 'Sex', 'Age'}));
```

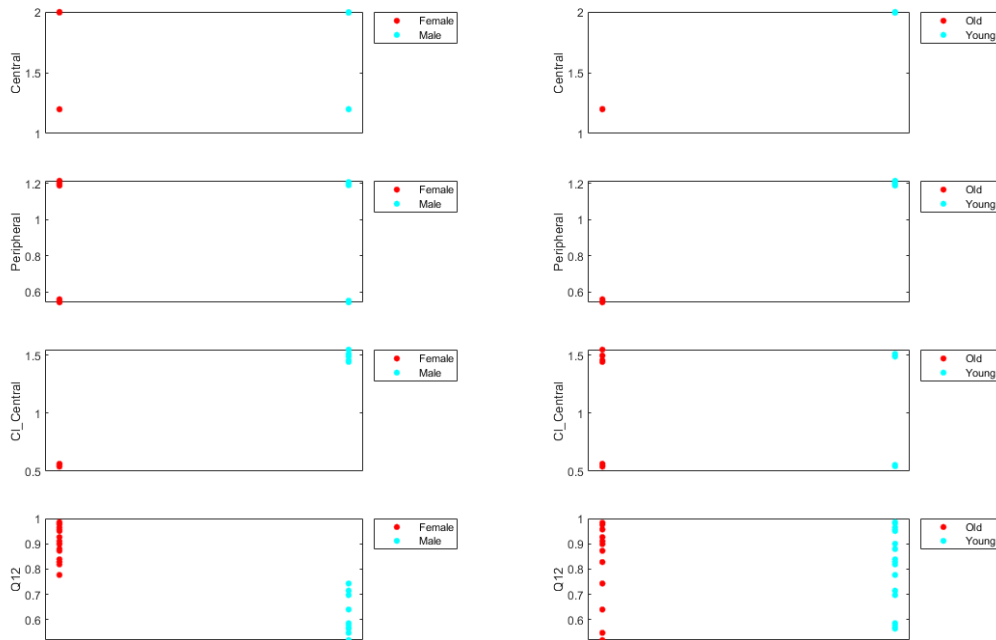
Add variables to the table containing each parameter's estimate.

```
allParamValues          = vertcat(unpooledFit.ParameterEstimates);
catParamValues.Central  = allParamValues.Estimate(strcmp(allParamValues.Name, 'Central'));
catParamValues.Peripheral = allParamValues.Estimate(strcmp(allParamValues.Name, 'Peripheral'));
catParamValues.Q12      = allParamValues.Estimate(strcmp(allParamValues.Name, 'Q12'));
catParamValues.Cl_Central = allParamValues.Estimate(strcmp(allParamValues.Name, 'Cl_Central'));
```

Plot estimates of each parameter for each category. `gscatter` requires Statistics and Machine Learning Toolbox™. If you do not have it, use other alternative plotting functions such as `plot`.

```
h          = figure;
ylabels    = {'Central', 'Peripheral', 'Cl_Central', 'Q12'};
plotNumber = 1;
for i = 1:4
    thisParam = estimatedParam(i).Name;

    % Plot for Sex category
    subplot(4,2,plotNumber);
    plotNumber = plotNumber + 1;
    gscatter(double(catParamValues.Sex), catParamValues.(thisParam), catParamValues.Sex);
    ax        = gca;
    ax.XTick  = [];
    ylabel(ylabels(i));
    legend('Location', 'bestoutside')
    % Plot for Age category
    subplot(4,2,plotNumber);
    plotNumber = plotNumber + 1;
    gscatter(double(catParamValues.Age), catParamValues.(thisParam), catParamValues.Age);
    ax        = gca;
    ax.XTick  = [];
    ylabel(ylabels(i));
    legend('Location', 'bestoutside')
end
% Resize the figure.
h.Position(:) = [100 100 1280 800];
```



Based on the plot, it seems that young individuals tend to have higher volumes of central and peripheral compartments (Central, Peripheral) than old individuals (that is, the volumes seem to be age-specific). In addition, males tend to have lower clearance rates (Cl\_Central) than females but the opposite for the Q12 parameter (that is, the clearance and Q12 seem to be sex-specific).

### Estimate Category-Specific Parameters

Use the 'CategoryVariableName' property of the estimatedInfo object to specify which category to use during fitting. Use 'Sex' as the group to fit for the clearance Cl\_Central and Q12 parameters. Use 'Age' as the group to fit for the Central and Peripheral parameters.

```
estimatedParam(1).CategoryVariableName = 'Age';
estimatedParam(2).CategoryVariableName = 'Age';
estimatedParam(3).CategoryVariableName = 'Sex';
```

```
estimatedParam(4).CategoryVariableName = 'Sex';
categoryFit = sbiofit(model,gData,responseMap,estimatedParam,dose)

categoryFit =
  OptimResults with properties:

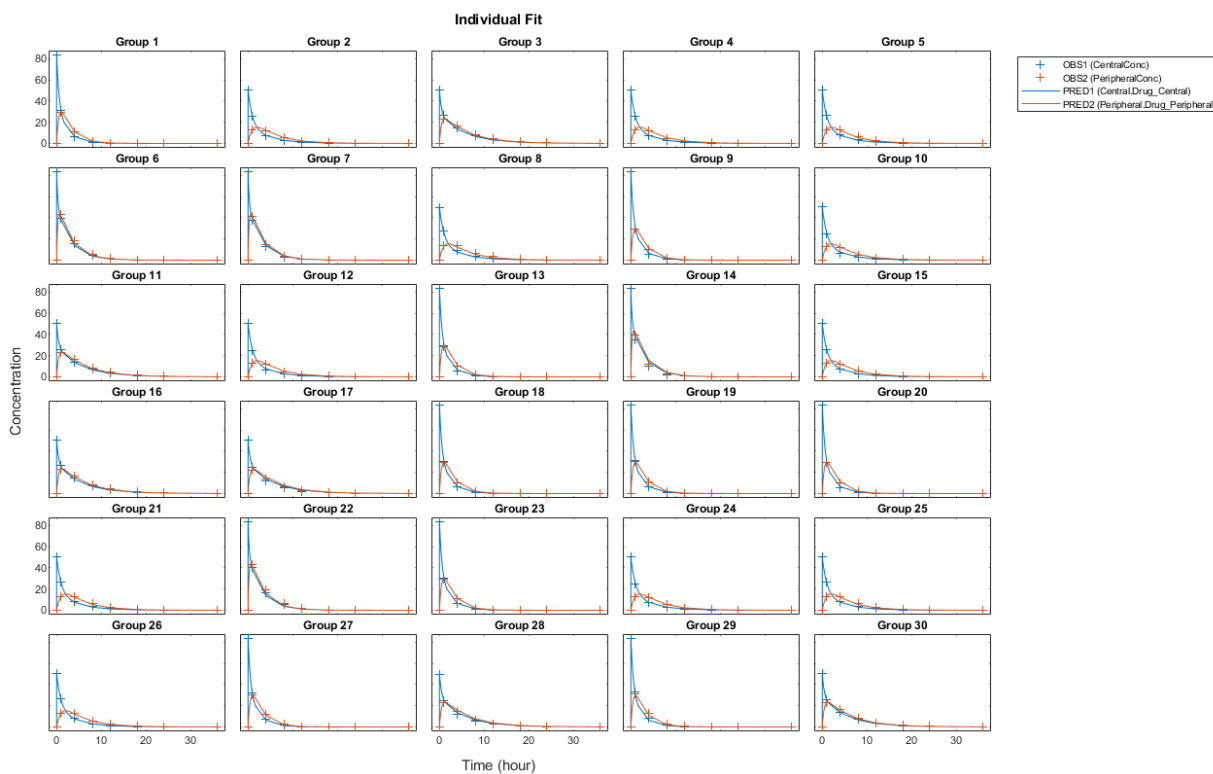
      ExitFlag: 3
      Output: [1x1 struct]
      GroupName: []
      Beta: [8x5 table]
      ParameterEstimates: [120x6 table]
      J: [240x8x2 double]
      COVB: [8x8 double]
      CovarianceMatrix: [8x8 double]
      R: [240x2 double]
      MSE: 0.4362
      SSE: 205.8690
      Weights: []
      LogLikelihood: -477.9195
      AIC: 971.8390
      BIC: 1.0052e+03
      DFE: 472
      DependentFiles: {1x3 cell}
      EstimatedParameterNames: {'Central' 'Peripheral' 'Q12' 'Cl_Central'}
      ErrorModelInfo: [1x3 table]
      EstimationFunction: 'lsqnonlin'
```

When fitting by category (or group), `sbiofit` always returns one results object, not one for each category level. This is because both male and female individuals are considered to be part of the same optimization using the same error model and error parameters, similarly for the young and old individuals.

### Plot Results

Plot the category-specific estimated results.

```
t = plot(categoryFit);
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```



For the `Cl_Central` and `Q12` parameters, all males had the same estimates, and similarly for the females. For the `Central` and `Peripheral` parameters, all young individuals had the same estimates, and similarly for the old individuals.

### Estimate Population-Wide Parameters

To better compare the results, fit the model to all of the data pooled together, that is, estimate one set of parameters for all individuals by setting the `'Pooled'` name-value pair argument to `true`. The warning message tells you that this option will ignore any category-specific information (if they exist).

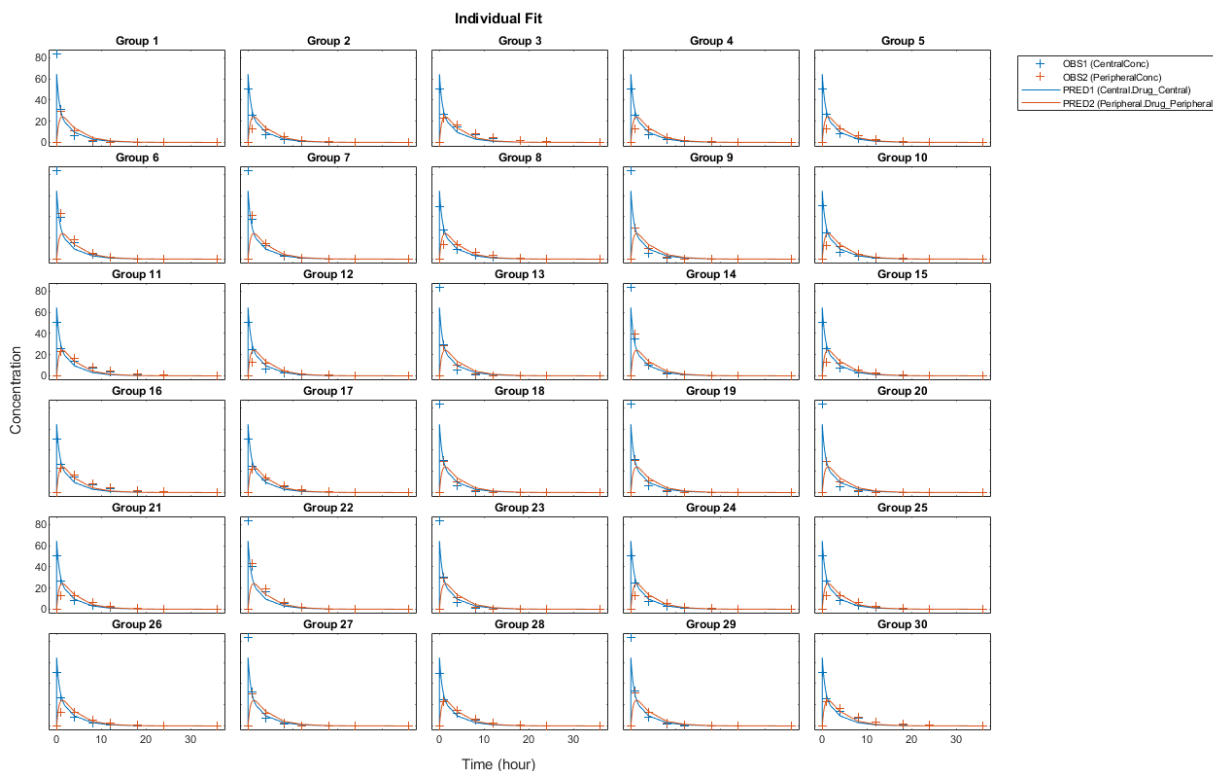
```
pooledFit = sbiofit(model,gData,responseMap,estimatedParam,dose,'Pooled',true);
```

Warning: CategoryVariableName property of the estimatedInfo object is ignored when using

## Plot Results

Plot the fitted results versus the original data. Although a separate plot was generated for each individual, the data was fitted using the same set of parameters (that is, all individuals had the same fitted line).

```
t = plot(pooledFit);
% Resize the figure.
t.hFig.Position(:) = [100 100 1280 800];
```



## Compare Residuals

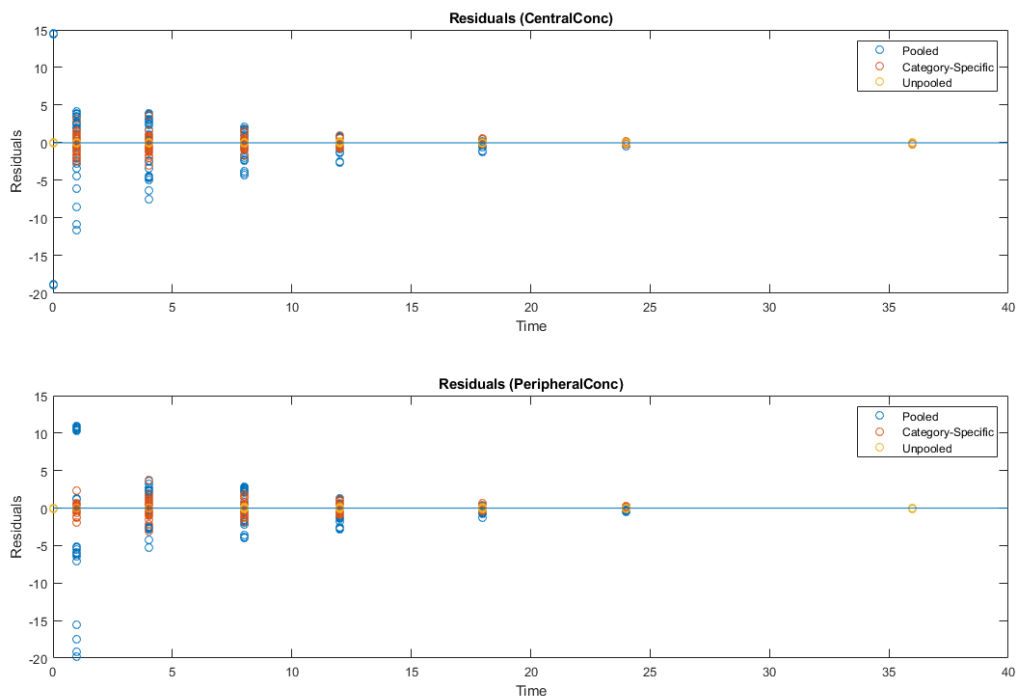
Compare residuals of CentralConc and PeripheralConc responses for each fit.

```
t = gData.Time;
allResid(:, :, 1) = pooledFit.R;
```



```
allResid(:,:,2) = categoryFit.R;
allResid(:,:,3) = vertcat(unpooledFit.R);

h = figure;
responseList = {'CentralConc', 'PeripheralConc'};
for i = 1:2
    subplot(2,1,i);
    oneResid = squeeze(allResid(:,i,:));
    plot(t,oneResid,'o');
    reffline(0,0); % A reference line representing a zero residual
    title(sprintf('Residuals (%s)', responseList{i}));
    xlabel('Time');
    ylabel('Residuals');
    legend({'Pooled', 'Category-Specific', 'Unpooled'});
end
% Resize the figure.
h.Position(:) = [100 100 1280 800];
```



As shown in the plot, the unpooled fit produced the best fit to the data as it fit the data to each individual. This was expected since it used the most number of degrees of freedom. The category-fit reduced the number of degrees of freedom by fitting the data to two categories (sex and age). As a result, the residuals were larger than the unpooled fit, but still smaller than the population-fit, which estimated just one set of parameters for all individuals. The category-fit might be a good compromise between the unpooled and pooled fitting provided that any hierarchical model exists within your data.

## See Also

`CovariateModel` object | `groupedData` object | `sbiofit` | `sbiofitmixed`

## Topics

“Fit a Two-Compartment Model to PK Profiles of Multiple Individuals” on page 1-69

“Estimate a Parameter from the Yeast G Protein Model” on page 1-89

**Introduced in R2014a**

## Event object

Store event information

### Description

Events are used to describe sudden changes in model behavior. An event lets you specify discrete transitions in model component values that occur when a user-specified condition become true. You can specify that the event occurs at a particular time, or specify a time-independent condition.

For details on how events are handled during a simulation, see “Events”.

See “Property Summary” on page 2-169 for links to event property reference pages.

Properties define the characteristics of an object. For example, an event object includes properties that allow you to specify the conditions to trigger an event (`Trigger`), and what to do after the event is triggered (`EventFcn`). Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the SimBiology desktop.

---

**Tip** If `UnitConversion` is on and your model has any event, follow the recommendation below.

Non-dimensionalize any parameters used in the event `trigger` if they are not already dimensionless. For example, suppose you have a trigger `x > 1`, where `x` is the species concentration in mole/liter. Non-dimensionalize `x` by scaling (dividing) it with a constant such as `x/x0 > 1`, where `x0` is a parameter defined as 1.0 mole/liter. Note that `x` does not have to have the same unit as the constant `x0`, but must be dimensionally consistent with it. For example, the unit of `x` can be picomole/liter instead of mole/liter.

---

### Constructor Summary

`addevent (model)`

Add event object to model object

## Method Summary

copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
get (any object)	Get object properties
set (any object)	Set object properties

## Property Summary

Active	Indicate object in use during simulation
EventFcns	Event expression
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Trigger	Event trigger
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

AbstractKineticLaw object, Configset object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object, Species object

**Introduced in R2007b**

## export (model)

Export SimBiology models for deployment and standalone applications

### Syntax

```
exportedModel = export(model)
exportedModel = export(model,editobjs)
exportedModel = export(model,editobjs,modifiers)
exportedModel = export(model,editobjs,editdoses,variants)
```

### Description

`exportedModel = export(model)` returns a `SimBiology.export.Model` object, `exportedModel`, from a SimBiology `model` object, `model` including all doses which are editable in the exported model. In addition, if the `model` has any active variants, they are automatically applied to determine the default initial values in the exported model. By default, all species, parameters, compartments, and doses are editable in the exported model. When you simulate the exported model, you can specify different initial values or different dose conditions.

`exportedModel = export(model,editobjs)` specifies `editobjs`, which is a species, parameter, compartment, or vector of these objects that are editable in the exported model. All doses are exported and are editable in the exported model. If the `model` has any active variants, they are automatically applied to determine the default initial values in the exported model. When you simulate the exported model, you can specify different initial values for `editobjs` or different dose conditions.

`exportedModel = export(model,editobjs,modifiers)` additionally specifies `modifiers` which is a dose, variant, vector of these objects or an empty array `[]`.

`exportedModel = export(model,editobjs,editdoses,variants)` additionally specifies `editdoses`, a dose object or vector of dose objects and `variants`, a variant object or vector of variant objects.

## Method Summary

Methods for exported model objects

<code>accelerate</code>	Prepare exported SimBiology model for acceleration
<code>getdose</code>	Return exported SimBiology model dose object
<code>getIndex</code>	Get indices into <code>ValueInfo</code> and <code>InitialValues</code> properties
<code>isAccelerated</code>	Determine whether an exported SimBiology model is accelerated
<code>simulate</code>	Simulate exported SimBiology model

## Input Arguments

### **model** — SimBiology model

SimBiology model object

SimBiology model, specified as a SimBiology model object.

### **editobj**s — Editable model quantities in the exported model

species object | parameter object | compartment object | vector of objects

Editable model quantities in the exported model, specified as a species, parameter, or compartment object or a vector of these objects.

### **modifiers** — Model modifiers

dose object | variant object | vector of objects | []

Model modifiers, specified as a dose or variant object, a vector of these objects, or an empty array [].

If `modifiers` is a vector of dose objects, then only these doses are editable in the exported model.

If `modifiers` is an empty array [], then no doses are editable in the exported model, and all active variants are applied to determine the default initial values of model quantities in the exported model.

If `modifiers` is a vector of variant objects, then specified variants are applied to determine the default initial values. All doses in the model are exported.

When you simulate the exported model, you can specify different initial values for `editobjs` or different dose conditions for editable doses.

### **editdoses — Editable doses**

dose object | vector of objects

Editable doses, specified as a dose object or vector of dose objects. The specified dose objects are editable in the exported model.

### **variants — Variants**

variant object | vector of objects

Variants, specified as a variant object or a vector of objects. The specified variant objects are applied to determine the default initial values in the exported model.

## Output Arguments

### **exportedModel — Exported model**

`SimBiology.export.Model`

Exported model, specified as a `SimBiology.export.Model` object.

## Examples

### **Export a SimBiology Model**

Export a `SimBiology` model object.

```
modelObj = sbmlimport('lotka');  
exportedModel = export(modelObj)
```

```
exportedModel =  
    Model with properties:  
  
        Name: 'lotka'  
    ExportTime: '27-Aug-2018 14:04:04'  
    ExportNotes: ''
```



Display the editable values (compartments, species, and parameters) information for the exported model object.

```
{exportedModel.ValueInfo.Name}

ans = 1x8 cell array
  Columns 1 through 7

    {'unnamed'}    {'x'}    {'y1'}    {'y2'}    {'z'}    {'c1'}    {'c2'}

  Column 8

    {'c3'}
```

There are 8 editable values in the exported model. Export the model again, allowing only the parameters (c1, c2, and c3) to be editable.

```
parameters = sbioselect(modelObj, 'Type', 'parameter');
exportedModelParam = export(modelObj, parameters);
{exportedModelParam.ValueInfo.Name}

ans = 1x3 cell array
    {'c1'}    {'c2'}    {'c3'}
```

Export the model a third time, allowing the parameters and species to be editable.

```
PS = sbioselect(modelObj, 'Type', {'species', 'parameter'});
exportedModelPS = export(modelObj, PS);
{exportedModelPS.ValueInfo.Name}

ans = 1x7 cell array
    {'x'}    {'y1'}    {'y2'}    {'z'}    {'c1'}    {'c2'}    {'c3'}
```

## See Also

Compartment object | Model object | Parameter object |  
SimBiology.export.Model | Species object

## Topics

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”

“Deploy a SimBiology Model”

**Introduced in R2012b**

## findUnusedComponents (model)

Find unused species, parameters, and compartments in a model

### Syntax

```
unused = findUnusedComponents(model)
unused = findUnusedComponents(model,dose)
unused = findUnusedComponents(model,dose,variant)
```

### Description

`unused = findUnusedComponents(model)` returns a vector of species, compartments, and parameters that are not used in `model`, which is a `SimBiology model` object. For details of what SimBiology checks to decide whether a component is used, see “Component Usage”.

`unused = findUnusedComponents(model,dose)` also searches for unused components in `dose`, which is a `RepeatDose` object, `ScheduleDose` object, or a vector of dose objects.

`unused = findUnusedComponents(model,dose,variant)` also searches for unused components in `variant`, which is a `variant` object or a vector of variant objects.

### Input Arguments

#### **model** — SimBiology model

SimBiology model object

SimBiology model, specified as a `SimBiology model` object.

#### **dose** — Dose object

`ScheduleDose` object | `RepeatDose` object | vector

Dose object, specified as a `ScheduleDose` object, `RepeatDose` object, or vector of dose objects.

### **variant — Variant**

variant object | vector

Variant, specified as a `variant` object or vector of variant objects.

## Output Arguments

### **unused — Unused components**

vector

Unused components, returned as a vector of species, parameters, and compartments.

## Examples

### **Find unused components in a model**

Load a sample project.

```
sbioloadproject gprotein.sbproj
```

Check if there is any unused species, compartments, or parameters.

```
unused = findUnusedComponents(m1)
```

```
unused =
```

```
    0x1 empty handle
```

Add some parameters to the model that are not used.

```
p1 = addparameter(m1, 'p1');  
p2 = addparameter(m1, 'p2');
```

Look for those unused parameters.

```
unused = findUnusedComponents(m1)
```

```
SimBiology Parameter Array
```

```
Index:      Name:      Value:      ValueUnits:
```

1	p1	1
2	p2	1

## See Also

Model object | findUsages(AbstractKineticLaw) |  
findUsages(species, parameter, compartment) |  
findUsages(unit, unitprefix)

## Topics

“Component Usage”

**Introduced in R2016b**

## findUsages (species, parameter, compartment)

Find out how a species, parameter, or compartment is used in a model

### Syntax

```
[componentList,usageTable] = findUsages(object)
[componentList,usageTable] = findUsages(object,dose)
[componentList,usageTable] = findUsages(object,dose,variant)
```

### Description

[componentList,usageTable] = findUsages(object) returns a vector of components that use the object and a table providing details about the usages. The object can be a species, parameter, or compartment object. For details of what SimBiology checks to decide whether a component is used, see “Component Usage”.

[componentList,usageTable] = findUsages(object,dose) also searches for usages of the object in dose, which is a RepeatDose object, ScheduleDose object, or a vector of dose objects.

[componentList,usageTable] = findUsages(object,dose,variant) also searches for usages of the object in variant, which is a variant object or a vector of variant objects.

### Input Arguments

**object — Species, parameter, compartment, unit, or unit prefix**

species object | parameter object | compartment object

Species, parameter, compartment, unit, or unit prefix, specified as a species object, parameter object, and compartment object.

**dose — Dose object**

ScheduleDose object | RepeatDose object | vector

Dose object, specified as a ScheduleDose object, RepeatDose object, or vector of dose objects.

**variant — Variant**

variant object | vector

Variant, specified as a variant object or vector of variant objects.

## Output Arguments

**componentList — List of model components that use the input object**

vector

List of model components that use the input object, returned as a vector.

**usageTable — Usage Information**

table

Usage information, returned as a table. Table variables are:

- *Component*- a vector of components that use the object
- *Property*- a cell array of character vectors listing the corresponding properties that refer to the object
- *Usage*- a cell array reporting the usages as follows:
  - For rules, the value of the Rule property,
  - For reactions, the value of the Reaction or ReactionRate property,
  - For kinetic laws, the name stored in the SpeciesVariableNames or ParameterVariableNames,
  - For events, the value of the Trigger property or the value of EventFcns{i}, where *i* the index of an event function that use the component.
  - For variants, the value of Content{i}, where *i* is the index of the content entry that use the component.
  - For doses, the value of the relevant property, that is, TargetName, DurationParameterName, or LagParameterName.

- For species making use of a compartment, the name of the compartment listed in the Parent property of the species.

## Examples

### Find out how a quantity is used in a model

Load a sample project.

```
sbioloadproject gprotein.sbproj
```

Check and see how the rate of G protein inactivation parameter kGd is used in the model.

```
kGd = sbioselect(m1, 'Name', 'kGd');  
[components, usages] = findUsages(kGd);
```

components is a vector of components that use the parameter kGd. Display these components.

```
for i = 1:length(components)  
    components(i)  
end
```

```
SimBiology Reaction Array
```

```
Index:    Reaction:  
1         Ga -> Gd
```

```
SimBiology Kinetic Law Array
```

```
Index:    KineticLawName:  
1         MassAction
```

Based on the information from the usages table, the parameter is being used as a reaction rate parameter.

```
usages
```



## See Also

compartment object | findUnusedComponents |  
findUsages(AbstractKineticLaw) | findUsages(unit,unitprefix) |  
parameter object | species object

## Topics

“Component Usage”

**Introduced in R2016b**

## findUsages (AbstractKineticLaw)

Find out how an AbstractKineticLaw object is used

### Syntax

```
rxnList= findUsages(aklObj,model)
```

### Description

`rxnList= findUsages(aklObj,model)` returns a vector of reactions in `model` that use the `AbstractKineticLaw` object `aklObj`. For details of what SimBiology checks to decide whether an abstract kinetic law is used, see “Component Usage”.

### Input Arguments

**aklObj** — Abstract kinetic law

AbstractKineticLaw object

Abstract kinetic law, specified as a `AbstractKineticLaw` object.

**model** — SimBiology model

model object | vector

SimBiology model, specified as a `model` object, or vector of model objects.

### Output Arguments

**rxnList** — List of reactions that use `aklObj`

vector

List of reactions that use `aklObj`, returned as a vector of reaction objects.

## Examples

### Find out how an abstract kinetic law is used in a model

Load a sample project.

```
sbioloadproject gprotein.sbproj
```

List all reactions in the model that use the MassAction abstract kinetic law.

```
akl = sbioselect('Type', 'abstract_kinetic_law', 'Name', 'MassAction');
list = findUsages(akl, m1)
```

```
SimBiology Reaction Array
```

```
Index:    Reaction:
1         L + R <-> RL
2         Gd + Gbg -> G
3         G + RL -> Ga + Gbg + RL
4         R <-> null
5         RL -> null
6         Ga -> Gd
```

## See Also

```
abstractkineticlaw object | findUnusedComponents |
findUsages(species, parameter, compartment) |
findUsages(unit, unitprefix) | kineticlaw object | reaction object
```

## Topics

“Component Usage”

**Introduced in R2016b**

## findUsages (unit, unitprefix)

Find out how a unit or unit prefix is used

### Syntax

```
[componentList,usageTable] = findUsages(obj)
[componentList,usageTable] = findUsages(obj,model)
[componentList,usageTable] = findUsages(obj,model,dose)
```

### Description

[componentList,usageTable] = findUsages(obj) returns a vector of components that use the unit or unit prefix object `obj` and a table providing details about how the `obj` is used in the `BuiltInLibrary` and `UserDefinedLibrary`. For details of what SimBiology checks to decide whether a unit or unit prefix is used by another component, see “Component Usage”.

[componentList,usageTable] = findUsages(obj,model) also searches for usages of the `obj` in `model`, which is a SimBiology model, or a vector of model objects.

[componentList,usageTable] = findUsages(obj,model,dose) also searches for usages of the `obj` in `dose`, which is a `ScheduleDose` object, `RepeatDose` object, or a vector of dose objects.

### Input Arguments

#### **obj** — Unit or unit prefix

unit object | unit prefix object

Unit or unit prefix, specified as a `unit` object, or `unitPrefix` object

#### **model** — SimBiology model

model object | vector

SimBiology model, specified as a `model` object or vector of model objects.

**dose — Dose object**

ScheduleDose object | RepeatDose object | vector

Dose object, specified as a ScheduleDose object, RepeatDose object, or vector of dose objects.

## Output Arguments

**componentList — List of model components that use the unit or unit prefix**

vector

List of model components that use the unit or unit prefix, returned as a vector.

**usageTable — Usage Information**

table

Usage information, returned as a table. Table variables are:

- *Component*- a vector of components that use the obj
- *Property*- a cell array of character vectors listing the corresponding properties that use the obj, and
- *Usage*- a cell array of character vectors stored in the relevant properties, that is, InitialAmountUnits, CapacityUnits, ValueUnits, TimeUnits, AmountUnits, RateUnits, or Composition.

## Examples

**Find out how a unit is used in a library**

Create the unit object.

```
gram = sbioselect('Type','Unit','Name','gram');
```

Check and see how the gram unit is used in the built-in library. If you have a custom library, the function also searches it.

```
gramUsage = findUsages(gram)
```

### SimBiology Unit Array

Index:	Library:	Name:	Composition:	Multiplier:
1	BuiltIn	gram	gram	1
2	BuiltIn	joule	(meter <sup>2</sup> *kilogra...	1
3	BuiltIn	calorie	(meter <sup>2</sup> *kilogra...	4.1868
4	BuiltIn	pascal	kilogram/(meter*...	1
5	BuiltIn	watt	(kilogram*meter <sup>...</sup>	1
6	BuiltIn	newton	(kilogram*meter)...	1
7	BuiltIn	dyne	(gram*centimeter...	1
8	BuiltIn	volt	(kilogram*meter <sup>...</sup>	1
9	BuiltIn	farad	(ampere <sup>2</sup> *second...	1
10	BuiltIn	ohm	(kilogram*meter <sup>...</sup>	1
11	BuiltIn	siemens	(ampere <sup>2</sup> *second...	1
12	BuiltIn	weber	(kilogram*meter <sup>...</sup>	1
13	BuiltIn	tesla	kilogram/(second...	1
14	BuiltIn	henry	(kilogram*meter <sup>...</sup>	1

## See Also

compartment object | findUnusedComponents |  
findUsages(AbstractKineticLaw) |  
findUsages(species,parameter,compartment) | parameter object | species  
object

## Topics

“Component Usage”

**Introduced in R2016b**

# fitted(LeastSquaresResults,OptimResults,NLINResults)

Return simulation results of SimBiology model fitted using least-squares regression

## Syntax

```
[yfit,parameterEstimates] = fitted(resultsObj)
```

## Description

[yfit,parameterEstimates] = fitted(resultsObj) returns simulation results yfit and parameter estimates parameterEstimates from a fitted SimBiology model.

---

**Tip** Use this method to retrieve simulation results from the fitted model if you did not specify the second optional output argument that corresponds to simulation results when you first ran `sbiofit`.

---

## Input Arguments

### resultsObj — Estimation results

OptimResults object | NLINResults object | vector of results objects

Estimation results, specified as an `OptimResults` object, `NLINResults` object, or vector of results objects which contains estimation results from running `sbiofit`.

## Output Arguments

### yfit — Simulation results

vector of SimData objects

Simulation results, returned as a vector of `SimData` objects. The states reported in yfit are the states that were included in the `responseMap` input argument of `sbiofit` as

well as any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the `SimBiology` model.

### **parameterEstimates — Estimated parameter values**

table

Estimated parameter values, returned as a table. This argument is identical to the `resultsObj.ParameterEstimates` property.

## Examples

### **Estimate a Parameter from the Yeast G Protein Model**

This example uses the yeast heterotrimeric G protein model and experimental data reported by [1]. For details about the model, see the **Background** section in “Parameter Scanning, Parameter Estimation, and Sensitivity Analysis in the Yeast Heterotrimeric G Protein Cycle”.

Load the G protein model.

```
sbioloadproject gprotein
```

Store the experimental data containing the time course for the fraction of active G protein.

```
time = [0 10 30 60 110 210 300 450 600]';  
GaFracExpt = [0 0.35 0.4 0.36 0.39 0.33 0.24 0.17 0.2]';
```

Create a `groupedData` object based on the experimental data.

```
tbl = table(time,GaFracExpt);  
grpData = groupedData(tbl);
```

Map the appropriate model component to the experimental data. In other words, indicate which species in the model corresponds to which response variable in the data. In this example, map the model parameter `GaFrac` to the experimental data variable `GaFracExpt` from `grpData`.

```
responseMap = 'GaFrac = GaFracExpt';
```



Use an `estimatedInfo` object to define the model parameter `kGd` as a parameter to be estimated.

```
estimatedParam = estimatedInfo('kGd');
```

Perform the parameter estimation.

```
fitResult = sbiofit(m1,grpData,responseMap,estimatedParam);
```

View the estimated parameter value of `kGd`.

```
fitResult.ParameterEstimates
```

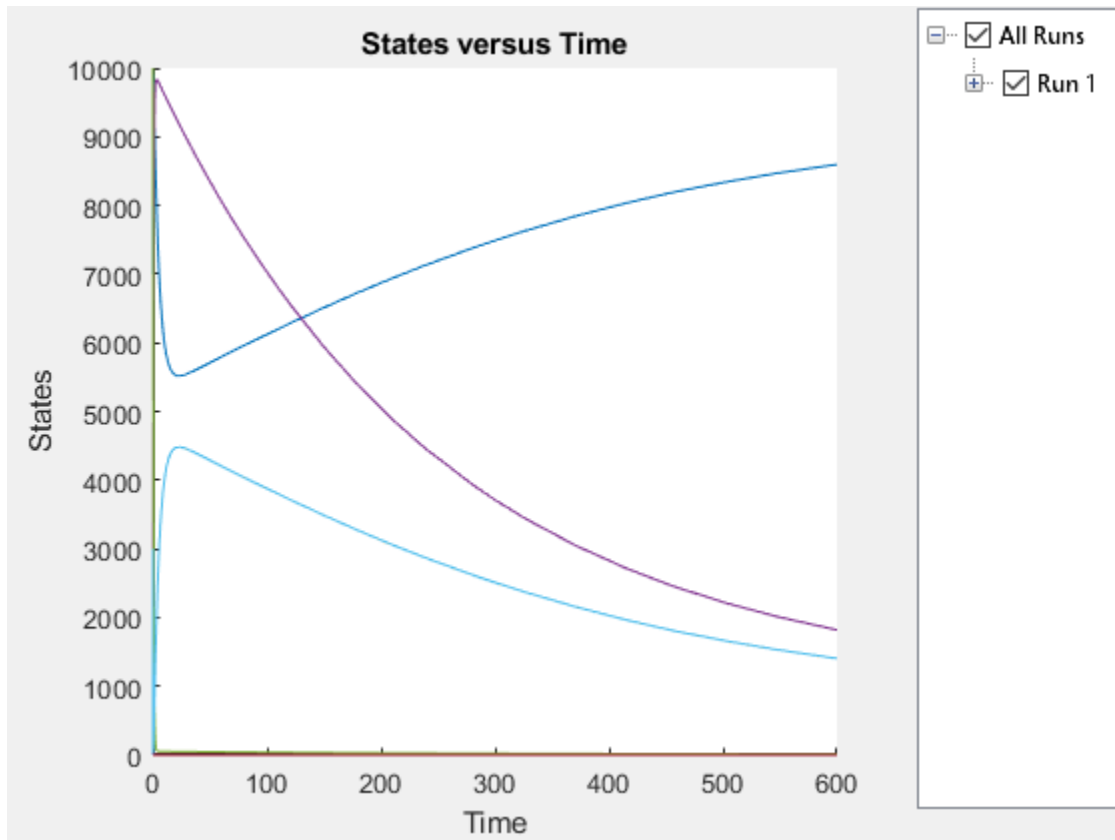
```
ans=1×3 table
      Name      Estimate      StandardError
      -----      -
      'kGd'      0.11307      3.4439e-05
```

Suppose you want to plot the model simulation results using the estimated parameter value. You can either rerun the `sbiofit` function and specify to return the optional second output argument, which contains simulation results, or use the `fitted` method to retrieve the results without rerunning `sbiofit`.

```
[yfit,paramEstim] = fitted(fitResult);
```

Plot the simulation results.

```
sbioplot(yfit);
```



## References

- [1] Yi, T-M., Kitano, H., and Simon, M. (2003). A quantitative characterization of the yeast heterotrimeric G protein cycle. PNAS. 100, 10764-10769.

## See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## fitted(NLMEResults)

Return the simulation results of a fitted nonlinear mixed-effects model

### Syntax

```
[yfit,parameterEstimates]= fitted(resultsObj)
[yfit,parameterEstimates]= fitted(resultsObj,'ParameterType',value)
```

### Description

[yfit,parameterEstimates]= fitted(resultsObj) returns simulation results yfit and parameter estimates parameterEstimates from a fitted nonlinear mixed-effect model.

[yfit,parameterEstimates]= fitted(resultsObj,'ParameterType',value) returns simulation results that are simulated using either individual or population parameter estimates. The two choices for value are 'population' or 'individual' (default).

---

**Tip** Use this method to retrieve simulation results from the fitted model if you did not specify the second or third optional output argument that corresponds to simulation results when you first ran sbiofitmixed.

---

### Input Arguments

**resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results returned by sbiofitmixed.

**value** — Parameter type

character vector | string

Parameter type, specified as 'population' or 'individual' (default). If 'population', the method returns the model simulation results using the population parameter estimates. If 'individual', it returns simulation results using the individual-specific parameter estimates.

## Output Arguments

### **yfit** — Simulation results

vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects. The states reported in `yfit` are the states that were included in the `responseMap` input argument of `sbiofitmixed` as well as any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the `SimBiology` model.

### **parameterEstimates** — Estimated parameter values

table

Estimated parameter values, returned as a table. This is identical to `resultsObj.IndividualParameterEstimates` property when the `value` argument is 'individual' or `resultsObj.PopulationParameterEstimates` property when the `value` is 'population'.

## See Also

`NLMEResults` object | `sbiofitmixed`

**Introduced in R2014a**

## get (any object)

Get object properties

### Syntax

```
objProperties = get(Obj)
PropertyValue = get(Obj, PropertyName)
PropertyValues = get(Obj, PropertyArray)
```

### Arguments

<i>PropertyValue</i>	Property value of <i>PropertyName</i> .
<i>Obj</i>	<ul style="list-style-type: none"> <li>Abstract kinetic law, compartment, configuration set, event, kinetic law, model, parameter, PKCompartment, PKData, PKModelDesign PKModelMap, reaction, rule, SimData, species, or variant object.</li> <li>Array of objects.</li> </ul>
<i>PropertyName</i>	Name of the property.
<i>objProperties</i>	Structure (or structure array) containing all properties and values of the object(s), <i>Obj</i> .
<i>PropertyArray</i>	Cell array of property names.
<i>PropertyValues</i>	<i>m</i> -by- <i>n</i> cell array, where <i>m</i> is the number of objects in <i>obj</i> and <i>n</i> is the number of property names in <i>PropertyArray</i> .

### Description

*objProperties* = `get(Obj)` returns all properties and property values of *Obj*.

*PropertyValue* = `get(Obj, PropertyName)` returns the value of specific property, *PropertyName*. Use quotes around the property name, for example, `get(model, 'Name')`. If *Obj* is an array of objects, *PropertyValue* is a *m*-by-1 cell array of property values, where *m* is the number of objects in *Obj*.

*PropertyValues* = `get(Obj,PropertyArray)` returns a cell array of property values for the properties in *PropertyArray*.

## Examples

### Get Model Dosing Information

Load the bioavailability model.

```
sbioloadproject('Bioavailability.sbproj');
```

Retrieve the name of the model.

```
modelName = get(m1,'Name')
```

```
modelName =  
'Bioavailability Model'
```

Check the dosing information.

```
m1.Doses
```

```
SimBiology Dose Array
```

Index:	Name:	Type:
1	Oral dose	schedule
2	IV Dose	schedule

Retrieve the *TimeUnits* and *AmountUnits* properties of the first (Oral) dose.

```
propValues = get(m1.Doses(1),{'TimeUnits','AmountUnits'})
```

```
propValues = 1x2 cell array  
    {'hour'}    {'milligram'}
```

Retrieve the properties of both Oral and IV doses.

```
propValues = get(m1.Doses,{'TimeUnits','AmountUnits'})
```

```
propValues = 2x2 cell array  
    {'hour'}    {'milligram'}
```

```
{'hour'} {'milligram'}
```

## See Also

getadjacencymatrix, getconfigset, getdata, getparameters, getsensmatrix, getspecies, getstoichmatrix, set

**Introduced in R2008b**

## getadjacencymatrix (model)

Get adjacency matrix from model object

### Syntax

```
M = getadjacencymatrix(modelObj)
[M, Headings] = getadjacencymatrix(modelObj)
[M, Headings, Mask] = getadjacencymatrix(modelObj)
```

### Arguments

M	Adjacency matrix for modelObj.
modelObj	Specify the model object.
Headings	Return row and column headings.  If species are in multiple compartments, species names are qualified with the compartment name in the form compartmentName.speciesName. For example, nucleus.DNA, cytoplasm.mRNA.
Mask	Return 1 for the species object and 0 for the reaction object to Mask.

### Description

`M = getadjacencymatrix(modelObj)` returns an adjacency matrix (M) for the model object (modelObj).

An adjacency matrix is defined by listing all species contained by modelObj and all reactions contained by modelObj column-wise and row-wise in a matrix. The reactants of the reactions are represented in the matrix with a 1 at the location of [row of species, column of reaction]. The products of the reactions are represented in the matrix with a 1 at the location of [row of reaction, column of species]. All other locations in the matrix are 0.



`[M, Headings] = getadjacencymatrix(modelObj)` returns the adjacency matrix to `M` and the row and column headings to `Headings`. `Headings` is defined by listing all `Name` property values of species contained by `modelObj` and all `Name` property values of reactions contained by `modelObj`.

`[M, Headings, Mask] = getadjacencymatrix(modelObj)` returns an array of 1s and 0s to `Mask`, where a 1 represents a species object and a 0 represents a reaction object.

## Examples

- 1 Read in `m1`, a model object, using `sbmlimport`:

```
m1 = sbmlimport('lotka.xml');
```

- 2 Get the adjacency matrix for `m1`:

```
[M, Headings] = getadjacencymatrix(m1)
```

## See Also

`getstoichmatrix`, `model` object

**Introduced in R2006a**

## getConfigset (model)

Get configuration set object from model object

### Syntax

```
configsetObj = getConfigset(modelObj, 'NameValue')  
configsetObj = getConfigset(modelObj)  
configsetObj = getConfigset(modelObj, 'active')
```

### Arguments

<i>modelObj</i>	Model object. Enter a variable name for a model object.
<i>NameValue</i>	Name of the configset object.
<i>configsetObj</i>	Object holding the simulation-specific information.

### Description

*configsetObj* = getConfigset(*modelObj*, 'NameValue') returns the configuration set attached to *modelObj* that is named *NameValue*, to *configsetObj*.

*configsetObj* = getConfigset(*modelObj*) returns a vector of all attached configuration sets, to *configsetObj*.

*configsetObj* = getConfigset(*modelObj*, 'active') retrieves the active configuration set.

A configuration set object stores simulation-specific information. A SimBiology model can contain multiple configsets with one being active at any given time. The active configuration set contains the settings that are used during the simulation.

Use the `setactiveconfigset` function to define the active configset. *modelObj* always contains at least one configset object with the name configured to 'default'. Additional configset objects can be added to *modelObj* with the method `addconfigset`.

## Examples

- 1 Retrieve the default configset object from the modelObj.

```
modelObj = sbiomodel('cell');
configsetObj = getconfigset(modelObj)
```

```
Configuration Settings - default (active)
```

```
SolverType:          ode15s
StopTime:            10
```

```
SolverOptions:
```

```
AbsoluteTolerance:  1.000000e-06
RelativeTolerance:  1.000000e-03
SensitivityAnalysis: false
```

```
RuntimeOptions:
```

```
StatesToLog:        all
```

```
CompileOptions:
```

```
UnitConversion:     false
DimensionalAnalysis: true
```

```
SensitivityAnalysisOptions:
```

```
Inputs:             0
Outputs:            0
```

- 2 Configure the SolverType to ssa.

```
set(configsetObj, 'SolverType', 'ssa')
get(configsetObj)
```

```
Active: 1
CompileOptions: [1x1 SimBiology.CompileOptions]
Name: 'default'
Notes: ''
RuntimeOptions: [1x1 SimBiology.RuntimeOptions]
SensitivityAnalysisOptions: [1x1 SimBiology.SensitivityAnalysisOptions]
SolverOptions: [1x1 SimBiology.SSASolverOptions]
SolverType: 'ssa'
StopTime: 10
MaximumNumberOfLogs: Inf
MaximumWallClock: Inf
TimeUnits: 'second'
Type: 'configset'
```

## **See Also**

model object, addconfigset, removeconfigset, setactiveconfigset

**Introduced in R2006a**

## getCovariateData (pkdata)

Create design matrix needed for fit

### Syntax

```
CovData = getCovariateData(PKDataObj)
```

### Description

*CovData* = getCovariateData(*PKDataObj*) creates *CovData*, a dataset array containing only the covariate data from the data set in *PKDataObj*, a PKData object. *CovData* contains one row for each individual and one column for each covariate.

---

**Tip** Use the getCovariateData method to view the covariate data when writing equations for the Expression on page 3-85 property of a CovariateModel on page 2-98 object.

---

### See Also

CovariateModel on page 2-98 | Expression on page 3-85 | PKData on page 2-275

### Topics

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”  
“Specify a Covariate Model”

**Introduced in R2011b**

## getdata (SimData)

Get data from SimData object array

### Syntax

```
[t, x, names] = getdata(simDataObj)
[Out] = getdata(simDataObj, 'FormatValue')
```

### Arguments

#### Output Arguments

<i>t</i>	An <i>n</i> -by-1 vector of time points.
<i>x</i>	An <i>n</i> -by- <i>m</i> data array. <i>t</i> and <i>names</i> label the rows and columns of <i>x</i> respectively.
<i>names</i>	An <i>m</i> -by-1 cell array of names.
<i>Metadata</i>	When used with the 'nummetadata' input argument, <i>Metadata</i> contains a cell array of metadata structures. The elements of <i>Metadata</i> label the columns of <i>x</i> .
<i>Out</i>	Data returned in the format specified in 'FormatValue', shown in "Input Arguments" on page 2-202. Depending on the specified 'FormatValue', <i>Out</i> contains one of the following: <ul style="list-style-type: none"> <li>• Structure array</li> <li>• SimData object</li> <li>• Time series object</li> <li>• Combined time series object from an array of SimData objects</li> </ul>

#### Input Arguments

<i>simDataObj</i>	SimData object. Enter a variable name for a SimData object.
-------------------	---

<i>FormatValue</i>	Choose a format from the following table.
<b>FormatValue</b>	<b>Description</b>
'num'	Specifies the format that lets you return data in numeric arrays. This is the default when <code>getdata</code> is called with multiple output arguments.
'nummetadata'	Specifies the format that lets you return a cell array of metadata structures in <i>metadata</i> instead of names. The elements of <i>metadata</i> label the columns of <i>x</i> .
'numqualnames'	Specifies the format that lets you return qualified names in <i>names</i> to resolve ambiguities.
'struct'	Specifies the format that lets you return a structure array holding both data and metadata. This is the default when you use a single output argument.
'simdata'	Specifies the format that lets you return data in a new SimData object. This format is more useful for SimData methods other than <code>getdata</code> .
'ts'	Specifies the format that lets you return data in time series objects, creating an individual time series for each state or column and SimData object in <code>simDataObj</code> .
'tslumped'	Specifies the format that lets you return data in time series objects, combining data from each SimData object into a single time series.

## Description

`[t, x, names] = getdata(simDataObj)` gets simulation time and state data from the SimData object `simDataObj`. When `simDataObj` contains more than one element,

the outputs *t*, *x*, *names* are cell arrays in which each cell contains data for the corresponding element of *simDataObj*.

[*Out*] = `getdata(simDataObj, 'FormatValue')` returns the data in the specified format. Valid formats are listed in “Input Arguments” on page 2-202.

## Examples

### Simulating and Retrieving Data

- 1 The project file, `radiodecay.sbproj`, contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace and simulate the model.

```
sbioloadproject('radiodecay');  
simDataObj = sbiosimulate(m1);
```

- 2 Get all the simulation data from the `SimData` object.

```
[t x names] = getdata(simDataObj);
```

### Retrieving Data for Ensemble Runs

- 1 The project file, `radiodecay.sbproj`, contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.

```
sbioloadproject('radiodecay');
```

- 2 Change the solver to use during the simulation and perform an ensemble run.

```
csObj = getconfigset(m1);  
set(csObj, 'SolverType', 'ssa');  
simDataObj = sbioensemblerun(m1, 10);
```

- 3 Get all the simulation data from the `SimData` object.

```
tsObj = getdata(simDataObj(1:5), 'ts');
```

## See Also

`display`, `get`, `resample`, `select`, `selectbyname`, `setactiveconfigset`

MATLAB function `struct`



**Introduced in R2007b**

## getdose

**Class:** SimBiology.export.Model

Return exported SimBiology model dose object

### Syntax

```
doses = getdose(model)
doses = getdose(model,doseName)
```

### Description

`doses = getdose(model)` returns all the `SimBiology.export.Dose` objects associated with the exported model.

`doses = getdose(model,doseName)` returns the export dose object with the `Name` property matching `doseName`.

### Input Arguments

**model**

`SimBiology.export.Model` object.

**doseName**

String containing a dose name to match against the `Name` property of the export dose objects in `model`.

**Default:** All dose objects.

## Output Arguments

### doses

Export dose objects in `model`, or the export dose object with Name property `doseName`.

## Examples

### Retrieve SimBiology Model Dose Objects

Open a sample SimBiology model project, and export the included model object.

```
sbioloadproject('AntibacterialPKPD')  
em = export(m1);
```

Display the editable doses in the exported model object.

```
doses = getdose(em)
```

```
doses =
```

```
1x4 RepeatDose array with properties:
```

```
Interval  
RepeatCount  
StartTime  
TimeUnits  
Amount  
AmountUnits  
DurationParameterName  
LagParameterName  
Name  
Notes  
Parent  
Rate  
RateUnits  
TargetName
```

The exported model has 4 repeated dose objects. Display the dose names.

```
{doses.Name}
```

```
ans =  
    '250 mg bid'    '250 mg tid'    '500 mg bid'    '500 mg tid'
```

Extract only the 3rd dose object from the exported model object.

```
dose3 = getdose(em, '500 mg bid')
```

```
dose3 =
```

```
RepeatDose with properties:
```

```
    Interval: 12  
    RepeatCount: 27  
    StartTime: 0  
    TimeUnits: 'hour'  
    Amount: 500  
    AmountUnits: 'milligram'  
    DurationParameterName: 'TDose'  
    LagParameterName: ''  
    Name: '500 mg bid'  
    Notes: ''  
    Parent: 'Antibacterial'  
    Rate: 0  
    RateUnits: ''  
    TargetName: 'Central.Drug'
```

## See Also

`SimBiology.export.Dose` | `SimBiology.export.Model` | `export`

## Topics

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”  
“Deploy a SimBiology Model”

## getdose (model)

Return SimBiology dose object

### Syntax

```
doseObj = getdose(modelObj)
doseObj = getdose(modelObj, 'DoseName')
```

### Arguments

<i>modelObj</i>	Selects a <code>model</code> object that contains a dose object.
<i>DoseName</i>	Name of a dose object contained in a model object. <i>DoseName</i> is from the dose object property, <code>Name</code> .

### Outputs

<i>doseObj</i>	ScheduleDose or RepeatDose object retrieved from a model object. A RepeatDose or ScheduleDose object defines an increase (dose) to a species amount during a simulation.
----------------	--

### Description

*doseObj* = `getdose(modelObj)` returns a Simbiology dose object (*doseObj*) contained in a Simbiology model object (*modelObj*).

*doseObj* = `getdose(modelObj, 'DoseName')` returns a SimBiology dose object (*modelObj*) with the name *DoseName*.

### Examples

Get a dose object from a model object.

- 1 Create a model object, and then add a dose object to the model object.

```
modelObj = sbiomodel('mymodel');  
doseObj = adddose(modelObj, 'dose1');
```

- 2 Get the dose object from a model object.

```
myModelDose = getdose(modelObj);
```

### See Also

Model methods:

- `adddose` — add a dose object to a model object
- `getdose` — get dose information from a model object
- `removedose` — remove a dose object from a model object

Dose object constructor `sbiodose`.

`ScheduleDose` object and `RepeatDose` object methods:

- `copyobj` — copy a dose object from one model object to another model object
- `get` — view properties for a dose object
- `set` — define or modify properties for a dose object

**Introduced in R2012b**

# getequations

Return system of equations for model object

## Syntax

```
equations = getequations(modelobj)
equations = getequations(modelobj,configsetobj,variantobj,doseobj)
```

## Description

`equations = getequations(modelobj)` returns `equations`, a character vector containing the system of equations that represent `modelobj`, a `model` object. The function uses any active `configset`, active variants, and active doses, if any, and generates the system of equations. You must specify a deterministic solver.

`equations = getequations(modelobj,configsetobj,variantobj,doseobj)` returns the system of equations that represent the model specified by a `Model` object, `Variant` objects, and dose objects (`RepeatDose` or `ScheduleDose`). The function uses only the specified `configset`, doses, and variants to generate the equations. Any other `configset`, doses, and variants are ignored. You must specify a deterministic solver.

If you set `csObj` to `[]`, then the function uses the active `configset` object.

If you set `variantObj` to `[]`, then the function uses no variants.

If you set `doseObj` to `[]`, then the function uses no doses.

## Input Arguments

### `modelObj`

Object of the `Model` on page 2-262 class.

**Note** If using `modelobj` as the only input argument, the active `Configset` object must specify a deterministic solver.

---

**Default:**

**configsetobj**

Object of the `Configset` on page 2-89 class. This object must specify a deterministic solver.

**Default:** [] (Empty, which specifies the active `Configset` object for `modelobj`)

**variantobj**

Object or array of objects of the `Variant` on page 2-504 class.

**Default:** [] (Empty, which specifies no variant object)

**doseobj**

Object or array of objects of the `RepeatDose` on page 2-405 or `ScheduleDose` on page 2-433 class.

**Default:** [] (Empty, which specifies no dose object)

## Output Arguments

**equations**

Character vector containing the system of equations that represent a model. Equations for reactions, rules, events, variants, and doses are included.

## Examples

### View System of Equations for Simple Model

View system of equations that represent a simple model, containing only reactions.



Import the lotka model, included with SimBiology, into a variable named `model1`:

```
model1 = sbmlimport('lotka');
```

View all equations that represent the `model1` model and its active configset:

```
mlequations = getequations(model1)

mlequations =

ODEs:
d(y1)/dt = 1/unnamed*(ReactionFlux1 - ReactionFlux2)
d(y2)/dt = 1/unnamed*(ReactionFlux2 - ReactionFlux3)
d(z)/dt = 1/unnamed*(ReactionFlux3)

Fluxes:
ReactionFlux1 = c1*y1*x
ReactionFlux2 = c2*y1*y2
ReactionFlux3 = c3*y2

Parameter Values:
c1 = 10
c2 = 0.01
c3 = 10
unnamed = 1

Initial Conditions:
x = 1
y1 = 900
y2 = 900
z = 0
```

MATLAB displays the ODEs, fluxes, parameter values, and initial conditions for the reactions in `model1`.

### View System of Equations for Model and Dose

View system of equations that represent a model, containing only reactions, and a repeated dose.

Import the lotka model, included with SimBiology, into a variable named `model1`:

```
model1 = sbmlimport('lotka');
```

Add a repeated dose to the model:

```
doseObj1 = adddose(model1, 'dose1', 'repeat');
```

Set the properties of the dose to administer 3 mg, at a rate of 10 mg/hour, 6 times, at an interval of every 24 hours, to species y1:

```
doseObj1.Amount = 0.003;  
doseObj1.AmountUnits = 'gram';  
doseObj1.Rate = 0.010;  
doseObj1.RateUnits = 'gram/hour';  
doseObj1.Repeat = 6;  
doseObj1.Interval = 24;  
doseObj1.TimeUnits = 'hour';  
doseObj1.TargetName = 'y1';
```

View all equations that represent the model1 model, its active configset, and the repeated dose:

```
m1_with_dose_equations = getequations (model1,[],[],doseObj1)
```

```
m1_with_dose_equations =
```

ODEs:

```
d(y1)/dt = 1/unnamed*(ReactionFlux1 - ReactionFlux2) + dose1  
d(y2)/dt = 1/unnamed*(ReactionFlux2 - ReactionFlux3)  
d(z)/dt = 1/unnamed*(ReactionFlux3)
```

Fluxes:

```
ReactionFlux1 = c1*y1*x  
ReactionFlux2 = c2*y1*y2  
ReactionFlux3 = c3*y2
```

Parameter Values:

```
c1 = 10  
c2 = 0.01  
c3 = 10  
unnamed = 1
```

Initial Conditions:

```
y1 = 900  
y2 = 900  
z = 0  
x = 1
```

Doses:	Type	Units
Variable		
dose1	repeatdose	gram

MATLAB displays the ODEs, fluxes, parameter values, and initial conditions for the reactions and the dose in `model1`.

## Tips

Use `getequations` to see the system of equations that represent a model for:

- Publishing purposes
- Model debugging

## See Also

`Configset` object | `Model` object | `RepeatDose` object | `ScheduleDose` object  
| `Variant` object

## Topics

“Equations View”

# getIndex

**Class:** SimBiology.export.Model

Get indices into ValueInfo and InitialValues properties

## Syntax

```
indices = getIndex(model,name)
indices = getIndex(model,name,type)
```

## Description

`indices = getIndex(model,name)` returns the indices of all ValueInfo objects in a SimBiology.export.Model object that have a QualifiedName or Name property that match the specified name input argument.

- `getIndex` first tries to match the QualifiedName property. If there are matches, then `getIndex` returns their indices.
- If there are no matches based on QualifiedName, then `getIndex` tries to match the Name property. If there are matches, then `getIndex` returns their indices.
- If there are no matches based on QualifiedName or Name, then `getIndex` returns [].

`indices = getIndex(model,name,type)` returns indices for only the ValueInfo objects with a Type property that matches the type input argument.

## Input Arguments

**model**

SimBiology.export.Model object.

**name**

Character vector containing a name to match against the `QualifiedName`, then `Name`, properties of the `ValueInfo` objects in `model`.

**type**

Character vector containing a name to match against the `Type` property of the `ValueInfo` objects in `model`.

**Default:** All types.

## Output Arguments

**indices**

Vector of indices indicating which `ValueInfo` objects in a `SimBiology.export.Model` object match on the specified name and type.

## Examples

### Index Exported SimBiology Editable Values

Load a sample SimBiology model object, and export.

```
modelObj = sbmlimport('lotka');  
em = export(modelObj);
```

Get the index of the editable value with name `y1`.

```
ix = getIndex(em, 'y1')
```

```
ix =
```

```
    3
```

Display the type of value.

```
em.ValueInfo(ix).Type
```

`ans =`

`species`

The name `y1` corresponds to an editable species.

### See Also

`SimBiology.export.Model` | `SimBiology.export.ValueInfo` | `export`

### Topics

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”

“Deploy a SimBiology Model”

## getparameters (kineticlaw)

Get specific parameters in kinetic law object

### Syntax

```
parameterObj = getparameters(kineticlawObj)
parameterObj = getparameters(kineticlawObj,
'ParameterVariablesValue')
```

### Arguments

<i>kineticlawObj</i>	Retrieve parameters used by the kinetic law object.
<i>ParameterVariablesValue</i>	Retrieve parameters used by the kinetic law object corresponding to the specified parameter in the ParameterVariables property of the kinetic law object.

### Description

*parameterObj* = getparameters(*kineticlawObj*) returns the parameters used by the kinetic law object *kineticlawObj* to *parameterObj*.

*parameterObj* = getparameters(*kineticlawObj*, 'ParameterVariablesValue') returns the parameter in the ParameterVariableNames property that corresponds to the parameter specified in the ParameterVariables property of *kineticlawObj*, to *parameterObj*. ParameterVariablesValue is the name of the parameter as it appears in the ParameterVariables property of *kineticlawObj*. ParameterVariablesValue can be a cell array of character vectors.

If you change the name of a parameter, you must configure all applicable elements such as rules that use the parameter, any user-specified ReactionRate, or the kinetic law

object property `ParameterVariableNames`. Use the method `setparameter` to configure `ParameterVariableNames`.

## Examples

Create a model, add a reaction, and assign the `ParameterVariableNames` for the reaction rate equation.

- 1 Create the model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object, of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

- 3 Add two parameter objects.

```
parameterObj1 = addparameter(kineticlawObj, 'Va');  
parameterObj2 = addparameter(kineticlawObj, 'Ka');
```

- 4 The 'Henri-Michaelis-Menten' kinetic law has two parameter variables (`Vm` and `Km`) that should to be set. To set these variables:

```
setparameter(kineticlawObj, 'Vm', 'Va');  
setparameter(kineticlawObj, 'Km', 'Ka');
```

- 5 To retrieve a parameter variable:

```
parameterObj3 = getparameters(kineticlawObj, 'Vm')
```

MATLAB returns:

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	Va	1	

```
parameterObj4 = getparameters (kineticlawObj, 'Km')
```

## See Also

`addparameter`, `getspecies`, `setparameter`



**Introduced in R2006a**

## getsensmatrix (SimData)

Get 3-D sensitivity matrix from SimData array

### Syntax

```
[T, R, Outputs, InputFactors] = getsensmatrix(simDataObj)
[T, R, Outputs, InputFactors] =
getsensmatrix(simDataObj, OutputNames, InputFactorNames)
```

### Arguments

<i>T</i>	<i>T</i> is an m-by-1 array specifying time points for the sensitivity data in R.
<i>R</i>	<i>R</i> is an m-by-n-by-p array of sensitivity data with times, outputs, and input factors corresponding to its first, second, and third dimensions respectively.  <i>R(:, i, j)</i> is the time course for the sensitivity of state <i>Outputs{i}</i> to the input factor <i>InputFactors{j}</i> .
<i>Outputs</i>	Name of the output factors, where output factors are the names of the states for which you want to calculate sensitivity.
<i>InputFactors</i>	Name of the input factors, where input factors are the names of the states with respect to which you want to calculate sensitivity.

### Description

`[T, R, Outputs, InputFactors] = getsensmatrix(simDataObj)` gets time and sensitivity data from the SimData object (*simDataObj*).

When *simDataObj* contains more than one element, the output arguments are cell arrays in which each cell contains data for the corresponding element of *simDataObj*.

The `getsensmatrix` method can only return sensitivity data that is contained in the SimData object. The sensitivity data that is logged in a SimData object is set at simulation

time by the configuration set used during the simulation. This is typically the model's active configuration set. For an explanation of how to set up a sensitivity calculation using the configuration set, see “Sensitivity Analysis”. Note in particular that the sensitivity data *R* returned by `getsensmatrix` may be normalized, as specified at simulation time.

```
[T, R, Outputs, InputFactors] =
getsensmatrix(simDataObj, OutputNames, InputFactorNames) gets sensitivity
data for the outputs specified by OutputNames and the input factors specified by
InputFactorNames.
```

*OutputNames* and *InputFactorNames* can both be any one of the following:

- Empty array
- Character vector or scalar string specifying a name
- String vector or cell array of character vectors specifying multiple names

Pass an empty array for *OutputNames* or *InputFactorNames* to ask for sensitivity data on all output factors or input factors contained in *simDataObj*, respectively. You can also use qualified names such as '*CompartmentName.SpeciesName*' or '*ReactionName.ParameterName*' to resolve ambiguities.

## Examples

This example shows how to retrieve sensitivity data from a SimData object.

### 1 Set up the simulation:

- a Import the radio decay model from SimBiology examples.

```
modelObj = sbmlimport('radiodecay');
```

- b Retrieve the configuration settings and the sensitivity analysis options from the `modelObj`.

```
configsetObj = getconfigset(modelObj);
sensitivityObj = get(configsetObj, 'SensitivityAnalysisOptions');
```

- c Specify the species for which you want sensitivity data in the `Outputs` property. All model species are selected in this example. Use the `sbioselect` function to retrieve the species objects from the model.

```
allSpeciesObj = sbioselect(modelObj, 'Type', 'species');  
set(sensitivityObj, 'Outputs', allSpeciesObj);
```

- d** Specify species and parameters with respect to which you want to calculate the sensitivities in the `Inputs` property.
- e** Enable `SensitivityAnalysis`.

```
set(configsetObj.SolverOptions, 'SensitivityAnalysis', true)  
get(configsetObj.SolverOptions, 'SensitivityAnalysis')
```

```
ans =
```

```
1
```

- f** Simulate and return the results in a `SimData` object.

```
simDataObj = sbiosimulate(modelObj)
```

- 2** Extract and plot sensitivity data from the `SimData` object.

- a** Use `getsensmatrix` to retrieve sensitivity data.

```
[t R outs ifacs] = getsensmatrix(simDataObj);
```

- b** Plot sensitivity values.

```
plot(t, R(:,:,2));  
legend(outs);  
title(['Sensitivities of species relative to ' ifacs{2}]);
```

## See Also

`display`, `get`, `getdata`, `resample`, `selectbyname`

MATLAB function struct

**Introduced in R2007b**

## getspecies (kineticlaw)

Get specific species in kinetic law object

### Syntax

```
speciesObj = getspecies(kineticLawObj)
speciesObj = getspecies(kineticLawObj, 'SpeciesVariablesValue')
```

### Arguments

<i>kineticLawObj</i>	Retrieve species used by the kinetic law object.
<i>SpeciesVariablesValue</i>	Retrieve species used by the kinetic law object corresponding to the specified species in the <i>SpeciesVariables</i> property of the kinetic law object.

### Description

*speciesObj* = *getspecies(kineticLawObj)* returns the species used by the kinetic law object *kineticLawObj* to *speciesObj*.

*speciesObj* = *getspecies(kineticLawObj, 'SpeciesVariablesValue')* returns the species in the *SpeciesVariableNames* property to *speciesObj*.

*SpeciesVariablesValue* is the name of the species as it appears in the *SpeciesVariables* property of *kineticLawObj*. *SpeciesVariablesValue* can be a cell array of character vectors.

Species names are referenced by reaction objects, kinetic law objects, and model objects. If you change the name of a species, the reaction updates to use the new name. You must, however, configure all other applicable elements such as rules that use the species, and the kinetic law object *SpeciesVariableNames*. Use the method *setspecies* to configure *SpeciesVariableNames*.

## Examples

Create a model, add a reaction, and then assign the `SpeciesVariableNames` for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object, of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

`reactionObj` `KineticLaw` property is configured to `kineticlawObj`.

- 3 The 'Henri-Michaelis-Menten' kinetic law has one species variable (S) that should be set. To set this variable:

```
setspecies(kineticlawObj, 'S', 'a');
```

- 4 Retrieve the species variable using `getspecies`.

```
speciesObj = getspecies (kineticlawObj, 'S')
```

MATLAB returns:

SimBiology Species Array

```
Index:  Compartment:  Name:  InitialAmount:  InitialAmountUnits:  
      1      unnamed      a           0
```

## See Also

`addspecies`, `getparameters`, `setParameter`, `setspecies`

**Introduced in R2006a**

## getstoichmatrix (model)

Get stoichiometry matrix from model object

### Syntax

```
M = getstoichmatrix(modelObj)
[M,objSpecies] = getstoichmatrix(modelObj)
[M,objSpecies,objReactions] = getstoichmatrix(modelObj)
```

### Arguments

<i>M</i>	Adjacency matrix for <i>modelObj</i> .
<i>modelObj</i>	Specify the model object.
<i>objSpecies</i>	Return the list of <i>modelObj</i> species by Name property of the species.  If the species are in multiple compartments, species names are qualified with the compartment name in the form <code>compartmentName.speciesName</code> . For example, <code>nucleus.DNA</code> , <code>cytoplasm.mRNA</code> .
<i>objReactions</i>	Return the list of <i>modelObj</i> reactions by the Name property of reactions.

### Description

`getstoichmatrix` returns a stoichiometry matrix for a model object.

`M = getstoichmatrix(modelObj)` returns a stoichiometry matrix for a SimBiology model object (*modelObj*) to *M*.

A stoichiometry matrix is defined by listing all reactions contained by *modelObj* column-wise and all species contained by *modelObj* row-wise in a matrix. The species of the

reaction are represented in the matrix with the stoichiometric value at the location of [row of species, column of reaction]. Reactants have negative values. Products have positive values. All other locations in the matrix are 0.

For example, if *modelObj* is a model object with two reactions with names R1 and R2 and Reaction values of  $2 A + B \rightarrow 3 C$  and  $B + 3 D \rightarrow 4 A$ , the stoichiometry matrix would be defined as:

	R1	R2
A	-2	4
B	-1	-1
C	3	0
D	0	-3

`[M,objSpecies] = getstoichmatrix(modelObj)` returns the stoichiometry matrix to *M* and the species to *objSpecies*. *objSpecies* is defined by listing all Name property values of species contained by *Obj*. In the above example, *objSpecies* would be {'A', 'B', 'C', 'D'};.

`[M,objSpecies,objReactions] = getstoichmatrix(modelObj)` returns the stoichiometry matrix to *M* and the reactions to *objReactions*. *objReactions* is defined by listing all Name property values of reactions contained by *modelObj*. In the above example, *objReactions* would be {'R1', 'R2'}.

## Examples

- 1 Read in *m1*, a model object, using `sbmlimport`:

```
m1 = sbmlimport('lotka.xml');
```

- 2 Get the stoichiometry matrix for the *m1*:

```
[M,objSpecies,objReactions] = getstoichmatrix(m1)
```

## See Also

`model` object, `getadjacencymatrix`, “Determining the Stoichiometry Matrix for a Model”



**Introduced in R2006a**

## getTable(ScheduleDose, RepeatDose)

Return data from SimBiology dose object as table

### Syntax

```
tbl = getTable(doseObj)
```

### Description

`tbl = getTable(doseObj)` returns dosing data from the dose object `doseObj` as a table `tbl`.

### Input Arguments

#### **doseObj** — Dose object

`ScheduleDose` object | `RepeatDose` object | array of dose objects

Dose object, specified as a `ScheduleDose` object or `RepeatDose` object or array of these objects.

### Output Arguments

#### **tbl** — Dosing data

table | cell array of tables

Dosing data, returned as a table or cell array of tables. If `doseObj` is an array of dose objects, then `tbl` is a cell array of tables with the same size as `doseObj`.

If `doseObj` is a `RepeatDose` object and any of the `StartTime`, `Amount`, `Rate`, `Interval`, and `RepeatCount` properties are parameterized, the table shows the name of the parameter in the corresponding column instead.

## Examples

### Retrieve a Table of Dosing Information from a RepeatDose Object

Create a RepeatDose object with some dosing information.

```
rdose = sbiodose('rd','repeat');
rdose.TargetName = 'x';
rdose.StartTime = 5;
rdose.TimeUnits = 'second';
rdose.Amount = 300;
rdose.AmountUnits = 'molecule';
rdose.Rate = 1;
rdose.RateUnits = 'molecule/second';
rdose.Interval = 100;
rdose.RepeatCount = 2;
```

Get a table of such dosing information.

```
tbl = getTable(rdose)
```

```
tbl =
```

StartTime	Amount	Rate	Interval	RepeatCount
5	300	1	100	2

Note that the units are also copied over and assigned to `tbl.Properties.VariableUnits` property.

```
tbl.Properties
```

```
ans =
```

```

    Description: ''
VariableDescriptions: {}
  VariableUnits: {'second' 'molecule' 'molecule/second' 'second' ''}
  DimensionNames: {'Row' 'Variable'}
        UserData: []
        RowNames: {}
  VariableNames: {'StartTime' 'Amount' 'Rate' 'Interval' 'RepeatCount'}
```

### Retrieve a Table of Dosing Information from a Schedule Object

Create a ScheduleDose object with some dosing information.

```
sdose = sbiodose('sdose', 'schedule');  
sdose.Amount = [100 200 300];  
sdose.Time = [5 10 15];
```

Get a table of such dosing information.

```
tbl = getTable(sdose)
```

```
tbl =
```

Time	Amount
5	100
10	200
15	300

### See Also

RepeatDose object | ScheduleDose object | setTable

**Introduced in R2014a**

## getvariant (model)

Get variant from model

### Syntax

```
variantObj = getvariant(modelObj)
```

```
variantObj = getvariant(modelObj, 'NameValue')
```

### Arguments

<i>variantObj</i>	Variant object returned by the <code>getvariant</code> method.
<i>modelObj</i>	Model object from which to get the variant.
'NameValue'	Name of the variant to get from the model object <i>modelObj</i> .

### Description

*variantObj* = `getvariant(modelObj)` returns SimBiology variant objects contained by the SimBiology model object *modelObj* to *variantObj*.

A SimBiology variant object stores alternate values for properties on a SimBiology model. For more information on variants, see `Variant` object.

*variantObj* = `getvariant(modelObj, 'NameValue')` returns the SimBiology variant object with the name *NameValue*, contained by the SimBiology model object, *modelObj*.

View properties for a variant object with the `get` command, and modify properties for a variant object with the `set` command.

**Note** Remember to use the `addcontent` method instead of using the `set` method on the Content property, because the `set` method replaces the data in the Content property whereas `addcontent` appends the data.

---

To copy a variant object to another model, use `copyobj`. To remove a variant object from a SimBiology model, use the `delete` method.

## Examples

- 1 Create a model containing several variants.

```
modelObj = sbiomodel('mymodel');
variantObj1 = addvariant(modelObj, 'v1');
variantObj2 = addvariant(modelObj, 'v2');
```

- 2 Get all variants in the model.

```
vObjs = getvariant(modelObj)
```

```
SimBiology Variant Array
```

Index:	Name:	Active:
1	v1	false
2	v2	false

- 3 Get the variant object named 'v2' from the model.

```
vObjv2 = getvariant(modelObj, 'v2');
```

## See Also

Model object, Variant object, `addvariant`, `removevariant`

**Introduced in R2007b**

# groupedData

Table-like collection of data and metadata

## Description

The `groupedData` object is the required data format to store data needed for fitting using `sbiofit` and `sbiofitmixed`. It is a table-like object that supports most of the functions and properties of `table` with a few differences.

- The `groupedData` object has two additional properties to identify the independent variable and an optional grouping variable.
- The `groupedData` object has two additional methods that let you create doses from data set containing dosing data and convert a `groupedData` object to a table.
- `groupedData.Properties` is a structure.

## Creation

## Syntax

```
grpData = groupedData  
grpData = groupedData(tbl)  
grpData = groupedData(tbl,groupVarName)  
grpData = groupedData(tbl,groupVarName,independentVarName)
```

## Description

`grpData = groupedData` creates an empty `groupedData` object.

`grpData = groupedData(tbl)` creates a `groupedData` object by copying a table or dataset object `tbl`. The `GroupVariableName` and `IndependentVariableName` properties of the `grpData` object are implicitly determined by looking for the first case-insensitive match to a list of variable names of `tbl` (`tbl.Properties.VariableNames`). For the grouping variable, the list of names is `ID`,

Group, I, and Run. For the independent variable, the list of names is Time, T, and IDV. If there are no matches, GroupVariableName and IndependentVariableName are set to empty character vectors ' '.

`grpData = groupedData(tbl,groupVarName)` sets the GroupVariableName property of the grpData object to groupVarName. The IndependentVariableName property is implicitly set as in the previous syntax.

`grpData = groupedData(tbl,groupVarName,independentVarName)` additionally sets the IndependentVariableName property of the grpData object to independentVarName.

### Input Arguments

#### **tbl — Data**

table | dataset

Data, specified as a table or dataset.

#### **groupVarName — Grouping variable name**

character vector | string

Grouping variable name, specified as a character vector or string. An empty character vector ' ' or string "" indicates that there is no group variable.

#### **independentVarName — Independent variable name**

character vector | string

Independent variable name, specified as a character vector or string. An empty character vector ' ' or string "" indicates that there is no independent variable.

### Output Arguments

#### **grpData — Grouped data**

groupedData object

Grouped data, returned as a groupedData object.



## Properties

The `groupedData` object supports all properties of `table` and provides the following additional properties.

### **GroupVariableName** — Name of grouping variable

character vector

Name of the grouping variable that indicates the groups in the data, specified as a character vector. To indicate that there are no groups (or just one group), set the property to the empty character vector `''`.

Example: `'ID'`

### **IndependentVariableName** — Name of independent variable

`''` (default) | character vector

Name of the independent variable in the data such as time, specified as a character vector.

Example: `'TIME'`

## Object Functions

The `groupedData` object supports most of the functions of `table` and provides the following additional functions.

`createDoses`      Create dose objects from `groupedData` object  
`groupedData2table`      Convert `groupedData` object to table

The `groupedData` object does not support these `table` functions: `addprop`, `addvars`, `convertvars`, `head`, `inner2outer`, `issortedrows`, `mergevars`, `movevars`, `removevars`, `rmprop`, `rows2vars`, `splitvars`, `tail`, and `topkrows`.

## Examples

### **Create groupedData from Existing Data Set**

Load the sample data set.

```
load pheno.mat ds
```

Create a `groupedData` object from the data set `ds`.

```
grpData = groupedData(ds);
```

Display the object properties.

```
grpData.Properties
```

```
ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Observations' 'Variables'}
    VariableNames: {'ID' 'TIME' 'DOSE' 'WEIGHT' 'APGAR' 'CONC'}
    VariableDescriptions: {}
    VariableUnits: {}
    VariableContinuity: []
    RowNames: {}
    CustomProperties: [1x1 matlab.tabular.CustomProperties]
    GroupVariableName: 'ID'
    IndependentVariableName: 'TIME'
```

`GroupVariableName` and `IndependentVariableName` have been automatically assigned to 'ID' and 'Time', respectively.

## See Also

[sbiofit](#) | [sbiofitmixed](#) | [table](#)

**Introduced in R2014a**

# groupedData2table

Convert groupedData object to table

## Syntax

```
tbl = groupedData2table(grpData)
```

## Description

`tbl = groupedData2table(grpData)` converts a `groupedData` object `grpData` to a table.

## Examples

### Convert groupedData to Table

Load the sample data set.

```
load pheno.mat ds
```

Create a `groupedData` object from the data set `ds`.

```
grpData = groupedData(ds);
```

Display the object properties.

```
grpData.Properties
```

```
ans = struct with fields:
    Description: ''
    UserData: []
    DimensionNames: {'Observations' 'Variables'}
    VariableNames: {'ID' 'TIME' 'DOSE' 'WEIGHT' 'APGAR' 'CONC'}
    VariableDescriptions: {}
    VariableUnits: {}
```

```
VariableContinuity: []  
    RowNames: {}  
    CustomProperties: [1x1 matlab.tabular.CustomProperties]  
    GroupVariableName: 'ID'  
    IndependentVariableName: 'TIME'
```

`GroupVariableName` and `IndependentVariableName` have been automatically assigned to 'ID' and 'Time', respectively.

Convert the `groupedData` to a table.

```
tbl = groupedData2table(grpData);
```

Display the first 5 rows of the table.

```
tbl(1:5,:)
```

```
ans=5x6 table  
    ID    TIME    DOSE    WEIGHT    APGAR    CONC  
    —    ———    ———    ————    ————    ———  
    1     0     25     1.4     7     NaN  
    1     2     NaN     1.4     7     17.3  
    1    12.5     3.5     1.4     7     NaN  
    1    24.5     3.5     1.4     7     NaN  
    1    37     3.5     1.4     7     NaN
```

## Input Arguments

### **grpData** — Grouped data

`groupedData` object

Grouped data, specified as a `groupedData` object.

## Output Arguments

### **tbl** — Data table

table

Data table, returned as a table.

## **See Also**

`groupedData` | `sbiofit` | `sbiofitmixed` | `table`

**Introduced in R2014a**

## isaccelerated(SimFunction)

Determine if SimFunction object is accelerated

### Syntax

```
tf = isaccelerated(F)  
tf = isaccelerated(F,computerType)
```

### Arguments

F	SimFunction object created by the createSimFunction method of a SimBiology model.
computerType	Character vector specifying a computer type. You can specify any valid archstr supported by the function computer.

### Description

tf = isaccelerated(F) returns true if SimFunction object F is accelerated for the current type of computer or false otherwise.

tf = isaccelerated(F,computerType) returns true if F is accelerated for the specified type of computer or false otherwise.

---

**Note** F is automatically accelerated at the first function execution. However, manually accelerate the object if you want it accelerated in your deployment applications.

---

### Examples

## Simulate SimFunction Object

This example uses the Lotka-Volterra (predator-prey) model described by Gillespie [1].

Load the sample project containing the lotka model.

```
sbioloadproject lotka;
```

Create a SimFunction object `f` with `c1` and `c2` as input parameters to be scanned, and `y1` and `y2` as the output of the function with no dose.

```
f = createSimFunction(m1,{'Reaction1.c1', 'Reaction2.c2'},{'y1', 'y2'}, [])
```

```
f =
```

```
SimFunction
```

```
Parameters:
```

Name	Value	Type
'Reaction1.c1'	10	'parameter'
'Reaction2.c2'	0.01	'parameter'

```
Observables:
```

Name	Type
'y1'	'species'
'y2'	'species'

```
Dosed: None
```

The `SimFunction` object `f` is not set for acceleration at the time of creation. But it will be automatically accelerated when executed.

```
f.isAccelerated
```

```
ans =
```

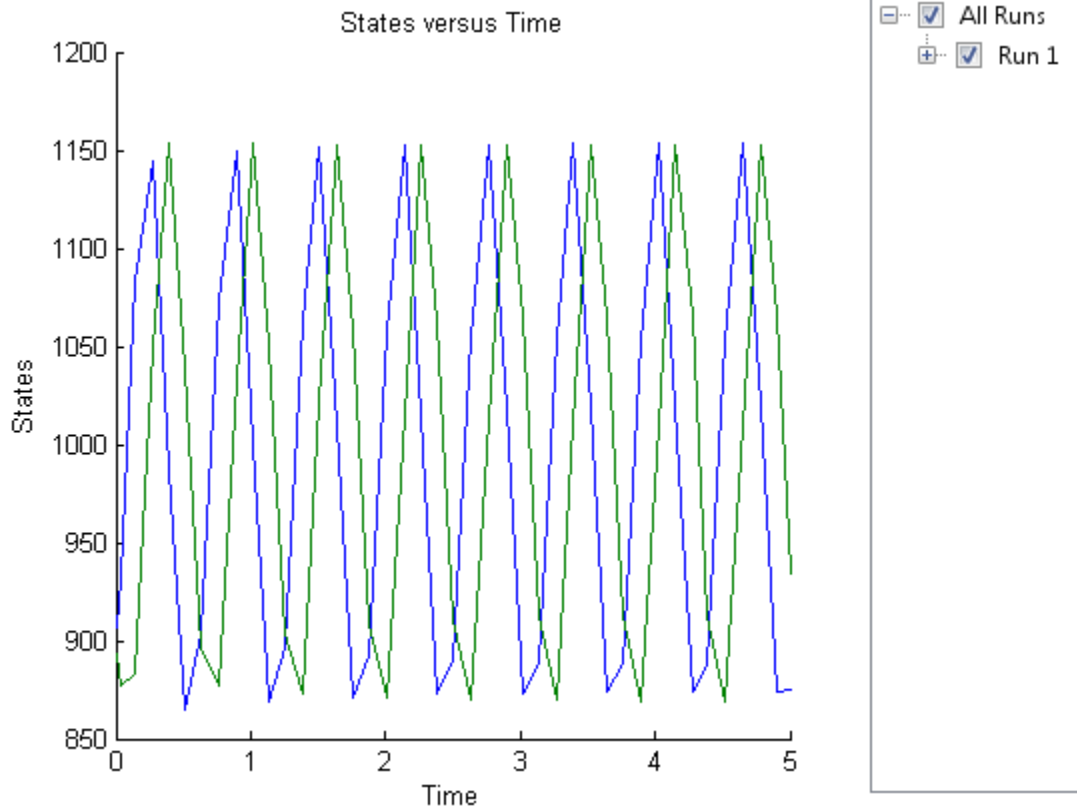
```
0
```

Define an input matrix that contains parameter values for `c1` and `c2`.

```
phi = [10 0.01];
```

Run simulations until the stop time is 5 and plot the simulation results.

```
sbioplot(f(phi,5))
```



Confirm the SimFunction object `f` was accelerated during execution.

```
f.isAccelerated
```



```
ans =
```

```
1
```

## See Also

`createSimFunction`, `SimFunction` object

## References

- [1] Gillespie D.T. "Exact Stochastic Simulation of Coupled Chemical Reactions," (1977) The Journal of Physical Chemistry, 81(25), 2340-2361.

**Introduced in R2012b**

## isAccelerated

**Class:** SimBiology.export.Model

Determine whether an exported SimBiology model is accelerated

### Syntax

```
tf = isAccelerated(model)
tf = isAccelerated(model, computerType)
```

### Description

`tf = isAccelerated(model)` returns `true` if `model` is accelerated for the current type of computer, and `false` otherwise.

`tf = isAccelerated(model, computerType)` returns `true` if `model` is accelerated for the specified computer type.

### Input Arguments

**model**

SimBiology.export.Model object.

**computerType**

String specifying a computer type. You can specify any valid `archstr` supported by the function `computer`.

## Output Arguments

**tf**

Logical value `true` if `model` is accelerated for the current computer type, or computer type specified by `computerType`. Logical value `false` if the exported model is not accelerated for the specified computer type.

## Examples

### Accelerate Exported SimBiology Model

Load a sample SimBiology model object, and export.

```
modelObj = sbmlimport('lotka');  
em = export(modelObj)
```

```
em =
```

```
Model with properties:
```

```
    Name: 'lotka'  
ExportTime: '12-Dec-2012 15:20:13'  
ExportNotes: ''
```

Accelerate the exported model.

```
accelerate(em);  
em.isAccelerated
```

```
ans =
```

```
1
```

The logical value `1` indicates that the exported model is accelerated.

## See Also

`SimBiology.export.Model` | `SimBiology.export.Model.accelerate` | `computer`

**Topics**

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”

“Deploy a SimBiology Model”

# KineticLaw object

Kinetic law information for reaction

## Description

The kinetic law object holds information about the abstract kinetic law applied to a reaction and provides a template for the reaction rate. In the model, the SimBiology software uses the information you provide in a fully defined kinetic law object to determine the `ReactionRate` property in the reaction object.

When you first create a kinetic law object, you must specify the name of the abstract kinetic law to use. The SimBiology software fills in the `KineticLawName` property and the `Expression` property in the kinetic law object with the name of the abstract kinetic law you specified and the mathematical expression respectively. The software also fills in the `ParameterVariables` property and the `SpeciesVariables` property of the kinetic law object with the values found in the corresponding properties of the abstract kinetic law object.

To obtain the reaction rate, you must fully define the kinetic law object:

- 1 In the `ParameterVariableNames` property, specify the parameters from the model that you want to substitute in the expression (`Expression` on page 3-88 property).
- 2 In the `SpeciesVariableNames` property, specify the species from the model that you want to substitute in the expression.

The SimBiology software substitutes in the expression, the names of parameter variables and species variables in the order specified in the `ParameterVariables` and `SpeciesVariables` properties respectively.

The software then shows the substituted expression as the reaction rate in the `ReactionRate` property of the reaction object. If the kinetic law object is not fully defined, the `ReactionRate` property remains ' ' (empty).

For links to kinetic law object property reference pages, see “Property Summary” on page 2-254.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can interactively change object properties in the SimBiology desktop.

For an explanation of how relevant properties relate to one another, see “Command Line” on page 2-250.

The following sections use a kinetic law example to show how you can fully define your kinetic law object to obtain the reaction rate in the SimBiology desktop and at the command line.

The Henri-Michaelis-Menten kinetic law is expressed as follows:

$$V_m * S / (K_m + S)$$

In the SimBiology software `Henri-Michaelis-Menten` is a built-in abstract kinetic law, where  $V_m$  and  $K_m$  are defined in the `ParameterVariables` property of the abstract kinetic law object, and  $S$  is defined in the `SpeciesVariables` property of the abstract kinetic law object.

## SimBiology Desktop

To fully define a kinetic law in the SimBiology desktop, define the names of the species variables and parameter variables that participate in the reaction rate.

## Command Line

To fully define the kinetic law object at the command line, define the names of the parameters in the `ParameterVariableNames` property of the kinetic law object, and define the species names in the `SpeciesVariableNames` property of the kinetic law object. For example, to apply the `Henri-Michaelis-Menten` abstract kinetic law to a reaction

```
A -> B
where Vm = Va, Km = Ka
and S = A
```

Define  $V_a$  and  $K_a$  in the `ParameterVariableNames` property to substitute the variables that are in the `ParameterVariables` property ( $V_m$  and  $K_m$ ). Define  $A$  in the `SpeciesVariableName` property to be used to substitute the species variable in the `SpeciesVariables` property ( $S$ ). Specify the order of the model parameters to be used

for substitution in the same order that the parameter variables are listed in the `ParameterVariables` property. Similarly, specify species order if more than one species variable is represented.

```
% Find the order of the parameter variables
% in the kinetic law expression.
```

```
get(kineticlawObj, 'ParameterVariables')
```

```
ans =
```

```
    'Vm'    'Km'
```

```
% Find the species variable in the
% kinetic law expression
```

```
get(kineticlawObj, 'SpeciesVariables')
```

```
ans =
```

```
    'S'
```

```
% Specify the parameters and species variables
% to be used in the substitution.
% Remember to specify order, for example Vm = Va
% Vm is listed first in 'ParameterVariables',
% therefore list Va first in 'ParameterVariableNames'.
```

```
set(kineticlawObj, 'ParameterVariableNames', {'Va' 'Ka'});
```

```
set(kineticlawObj, 'SpeciesVariableNames', {'A'});
```

The rate equation is assigned in the reaction object as follows:

$$V_a * A / (K_a + A)$$

For a detailed procedure, see “Examples” on page 2-254.

The following table summarizes the relationships between the properties in the abstract kinetic law object and the kinetic law object in the context of the above example.

Property	Property Purpose	Abstract Kinetic Law Object	Kinetic Law Object
Name (abstract kinetic law object) KineticLawName (kinetic law object)	Name of abstract kinetic law applied to a reaction. For example:  Henri-Michaelis-Menten	Read-only for built-in abstract kinetic law. User-determined for user-defined abstract kinetic law.	Read-only
Expression	Mathematical expression used to determine the reaction rate equation.  For example:  $V_m * S / (K_m + S)$	Read-only for built-in abstract kinetic law. User-determined for user-defined abstract kinetic law.	Read-only; depends on abstract kinetic law applied to reaction.
ParameterVariables	Variables in Expression that are parameters. For example:  Vm and Km	Read-only for built-in abstract kinetic law. User-determined for user-defined abstract kinetic law.	Read-only; depends on abstract kinetic law applied to reaction.
SpeciesVariables	Variables in Expression that are species. For example:  S	Read-only for built-in abstract kinetic law. User-determined for user-defined abstract kinetic law.	Read-only; depends on abstract kinetic law applied to reaction.
ParameterVariableNames	Variables in ReactionRate that are parameters. For example:  Va and Ka	Not applicable	Define these variables corresponding to ParameterVariables.



Property	Property Purpose	Abstract Kinetic Law Object	Kinetic Law Object
SpeciesVariablesNames	Variables in ReactionRate that are species. For example: A	Not applicable	Define these variables corresponding to SpeciesVariables.

## Constructor Summary

addkineticlaw (reaction)      Create kinetic law object and add to reaction object

## Method Summary

addparameter (model, kineticlaw)      Create parameter object and add to model or kinetic law object

copyobj (any object)      Copy SimBiology object and its children

delete (any object)      Delete SimBiology object

display (any object)      Display summary of SimBiology object

get (any object)      Get object properties

getparameters (kineticlaw)      Get specific parameters in kinetic law object

getspecies (kineticlaw)      Get specific species in kinetic law object

reorder (model, compartment, kinetic law)      Reorder component lists

set (any object)      Set object properties

setparameter (kineticlaw)      Specify specific parameters in kinetic law object

setspecies (kineticlaw)      Specify species in kinetic law object

## Property Summary

Expression (AbstractKineticLaw, KineticLaw)	Expression to determine reaction rate equation
KineticLawName	Name of kinetic law applied to reaction
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parameters	Array of parameter objects
ParameterVariableNames	Cell array of reaction rate parameters
ParameterVariables	Parameters in kinetic law definition
Parent	Indicate parent object
SpeciesVariableNames	Cell array of species in reaction rate equation
SpeciesVariables	Species in abstract kinetic law
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

This example shows how to define the reaction rate for a reaction.

- 1 Create a model object, and add a reaction object to the model.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'A -> B');
```

- 2 Define a kinetic law for the reaction object.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

- 3 Query the parameters and species variables defined in the kinetic law.

```
get(kineticlawObj, 'ParameterVariables')
```

```
ans =
```

```
    'Vm'    'Km'
```

```
get(kineticlawObj, 'SpeciesVariables')
ans =
```

```
    'S'
```

- 4 Define Va and Ka as ParameterVariableNames, which correspond to the ParameterVariables Vm and Km. To set these variables, first create the parameter variables as parameter objects (parameterObj1, parameterObj2) with the names Va and Ka, and then add them to kineticlawObj. The species object with Name A is created when reactionObj is created and need not be redefined.

```
parameterObj1 = addparameter(kineticlawObj, 'Va');
parameterObj2 = addparameter(kineticlawObj, 'Ka');
```

- 5 Set the variable names for the kinetic law object.

```
set(kineticlawObj, 'ParameterVariableNames', {'Va' 'Ka'});
set(kineticlawObj, 'SpeciesVariableNames', {'A'});
```

- 6 Verify that the reaction rate is expressed correctly in the reaction object ReactionRate property.

```
get (reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =
```

```
Va*A/(Ka+A)
```

## See Also

AbstractKineticLaw object, Configset object, Model object, Parameter object, Reaction object, Root object, Rule object, Species object

SimBiology property Expression(AbstractKineticLaw, KineticLaw)

### Introduced in R2006b

## LeastSquaresResults object

Results object containing estimation results from least-squares regression

### Description

The `LeastSquaresResults` object is a superclass of two results objects: `NLINResults` object and `OptimResults` object. These objects contain estimation results from fitting a SimBiology model to data using `sbiofit` with any supported algorithm.

If `sbiofit` uses the `nlinfit` estimation algorithm, the results object is the `NLINResults` object. If `sbiofit` uses any other supporting algorithm, then the results object is an `OptimResults` object. See the `sbiofit` function for the list of supported algorithms.

## Method Summary

`boxplot(LeastSquaresResults, OptimResults, NLINResults)`

Create box plot showing the variation of estimated SimBiology model parameters

`fitted(LeastSquaresResults, OptimResults, NLINResults)`

Return simulation results of SimBiology model fitted using least-squares regression

`plot(LeastSquaresResults, OptimResults, NLINResults)`

Compare simulation results to the training data, creating a time-course subplot for each group

`plotActualVersusPredicted(LeastSquaresResults, OptimResults, NLINResults)`

Compare predictions to actual data, creating a subplot for each response

`plotResiduals(LeastSquaresResults, OptimResults, NLINResults)`

Plot residuals for each response, using time.

## Property Summary

GroupName	Categorical variable representing the name of the group associated with the results, or [] if the 'Pooled' name-value pair argument was set to true when you ran <code>sbiofit</code> .
Beta	<p>Table of estimated parameters where the <math>j</math>th row represents the <math>j</math>th estimated parameter <math>\beta_j</math>. It contains transformed values of parameter estimates if any parameter transform is specified.</p> <p>Standard errors of these parameter estimates (<code>StandardError</code>) are calculated as:  <code>sqrt(diag(COVB))</code>.</p> <p>It can also contain the following variables:</p> <ul style="list-style-type: none"> <li>• <code>Bounds</code> — the values of transformed parameter bounds that you specified during fitting</li> <li>• <code>CategoryVariableName</code> — the names of categories or groups that you specified during fitting</li> <li>• <code>CategoryValue</code> — the values of category variables specified by <code>CategoryVariableName</code></li> </ul> <p>This table contains one row per distinct parameter value.</p>

ParameterEstimates	<p>Table of estimated parameters where the <math>j</math>th row represents the <math>j</math>th estimated parameter <math>\beta_j</math>. This table contains untransformed values of parameter estimates.</p> <p>Standard errors of these parameter estimates (StandardError) are calculated as:  <math>\text{sqrt}(\text{diag}(\text{CovarianceMatrix}))</math>.</p> <p>It can also contain the following variables:</p> <ul style="list-style-type: none"> <li>• <b>Bounds</b> — the values of parameter bounds that you specified during fitting</li> <li>• <b>CategoryVariableName</b> — the names of categories or groups that you specified during fitting</li> <li>• <b>CategoryValue</b> — the values of category variables specified by CategoryVariableName</li> </ul> <p>This table contains sets of parameter values that are identified for each individual or group.</p>
J	<p>Jacobian matrix of the model, with respect to an estimated parameter, that is,</p> $J(i, j, k) = \left. \frac{\partial y_k}{\partial \beta_j} \right _{t_i}$ <p>where <math>t_i</math> is the <math>i</math>th time point, <math>\beta_j</math> is the <math>j</math>th estimated parameter in the transformed space, and <math>y_k</math> is the <math>k</math>th response in the group of data.</p>
COVB	<p>Estimated covariance matrix for Beta, which is calculated as: <math>\text{COVB} = \text{inv}(J' * J) * \text{MSE}</math>.</p>

CovarianceMatrix	<p>Estimated covariance matrix for ParameterEstimates, which is calculated as: <math>\text{CovarianceMatrix} = \text{T}' * \text{COVB} * \text{T}</math>, where <math>\text{T} = \text{diag}(\text{JInvT}(\text{Beta}))</math>.</p> <p><math>\text{JInvT}(\text{Beta})</math> returns a Jacobian matrix of Beta which is inverse transformed accordingly if you specified any transform to estimated parameters.</p> <p>For instance, suppose you specified the log-transform for an estimated parameter <math>x</math> when you ran <code>sbiofit</code>. The inverse transform is: <math>\text{InvT} = \exp(x)</math>, and its Jacobian is: <math>\text{JInvT} = \exp(x)</math> since the derivative of <math>\exp</math> is also <math>\exp</math>.</p>
R	Residuals matrix where $R_{ij}$ is the residual for the $i$ th time point and the $j$ th response in the group of data.
LogLikelihood	Maximized loglikelihood for the fitted model.
AIC	Akaike Information Criterion (AIC), calculated as $\text{AIC} = 2 * (-\text{LogLikelihood} + P)$ , where $P$ is the number of parameters.
BIC	Bayes Information Criterion (BIC), calculated as $\text{BIC} = -2 * \text{LogLikelihood} + P * \log(N)$ , where $N$ is the number of observations, and $P$ is the number of parameters.
DFE	Degrees of freedom for error, calculated as $\text{DFE} = N - P$ , where $N$ is the number of observations and $P$ is the number of parameters.
MSE	Mean squared error.
SSE	Sum of squared (weighted) errors or residuals.
Weights	Matrix of weights with one column per response and one row per observation.
EstimatedParameterNames	Cell array of character vectors specifying estimated parameter names.



ErrorModelInfo	<p>Table describing the error models and estimated error model parameters.</p> <ul style="list-style-type: none"> <li>• It has one row per error model.</li> <li>• The <code>ErrorModelInfo.Properties.RowsNames</code> property identifies which responses the row applies to.</li> <li>• The table contains three variables: <code>ErrorModel</code>, <code>a</code>, and <code>b</code>. The <code>ErrorModel</code> variable is categorical. The variables <code>a</code> and <code>b</code> can be <code>NaN</code> when they do not apply to a particular error model.</li> </ul> <p>There are four built-in error models. Each model defines the error using a standard mean-zero and unit-variance (Gaussian) variable <math>e</math>, the function value <math>f</math>, and one or two parameters <math>a</math> and <math>b</math>. In SimBiology, the function <math>f</math> represents simulation results from a SimBiology model.</p> <ul style="list-style-type: none"> <li>• 'constant': <math>y = f + ae</math></li> <li>• 'proportional': <math>y = f + b f e</math></li> <li>• 'combined': <math>y = f + (a + b f )e</math></li> <li>• 'exponential': <math>y = f * \exp(ae)</math></li> </ul>
EstimationFunction	Name of the estimation function.
DependentFiles	File names to include for deployment.

**Note** Loglikelihood, AIC, and BIC properties are empty for `LeastSquaresResults` objects that were obtained before R2016a.

## See Also

`NLINResults` object | `OptimResults` object | `sbiofit` | `sbiofitmixed`

**Introduced in R2014a**

## Model object

Model and component information

### Description

The SimBiology model object represents a *model*, which is a collection of interrelated reactions and rules that transform, transport, and bind species. The model includes model components such as compartments, reactions, parameters, rules, and events. Each of the components is represented as a property of the model object. A model object also has a default configuration set object to define simulation settings. You can also add more configuration set objects to a model object.

See “Property Summary” on page 2-264 for links to model property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the SimBiology desktop.

You can retrieve SimBiology model objects from the SimBiology root object. A SimBiology model object has its `Parent` property set to the SimBiology root object. The root object contains a list of model objects that are accessible from the MATLAB command line and from the SimBiology desktop. Because both the command line and the desktop point to the same model object in the `Root` object, any changes you make to the model at the command line are reflected in the desktop, and vice versa.

### Constructor Summary

`sbiomodel`

Construct model object

## Method Summary

addcompartment (model, compartment)	Create compartment object
addconfigset (model)	Create configuration set object and add to model object
adddose (model)	Add dose object to model
addevent (model)	Add event object to model object
addparameter (model, kineticlaw)	Create parameter object and add to model or kinetic law object
addreaction (model)	Create reaction object and add to model object
addrule (model)	Create rule object and add to model object
addspecies (model, compartment)	Create species object and add to compartment object within model object
addvariant (model)	Add variant to model
copyobj (any object)	Copy SimBiology object and its children
createSimFunction (model)	Create SimFunction object
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
export (model)	Export SimBiology models for deployment and standalone applications
findUnusedComponents (model)	Find unused species, parameters, and compartments in a model
get (any object)	Get object properties
getadjacencymatrix (model)	Get adjacency matrix from model object
getconfigset (model)	Get configuration set object from model object
getdose (model)	Return SimBiology dose object
getequations	Return system of equations for model object
getstoichmatrix (model)	Get stoichiometry matrix from model object
getvariant (model)	Get variant from model
removeconfigset (model)	Remove configuration set from model
removedose (model)	Add dose object to model
removevariant (model)	Remove variant from model
reorder (model, compartment, kinetic law)	Reorder component lists
set (any object)	Set object properties

## Property Summary

Compartments	Array of compartments in model or compartment
Events	Contain all event objects
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parameters	Array of parameter objects
Parent	Indicate parent object
Reactions	Array of reaction objects
Rules	Array of rules in model object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

AbstractKineticLaw object, Configset object, KineticLaw object, Parameter object, Reaction object, Root object, Rule object, Species object

**Introduced in R2006b**

# NLINResults object

Estimation results object, subclass of `LeastSquaresResults`

## Description

The `NLINResults` object contains estimation results from fitting a SimBiology model to data using `sbiofit` with `nlinfit` as a choice of estimation algorithm. See the `sbiofit` function for a list of other supported algorithms.

## Method Summary

The `NLINResults` object shares all methods of the `LeastSquaresResults` object.

## Property Summary

The `NLINResults` object shares all properties of the `LeastSquaresResults` object.

## See Also

`LeastSquaresResults` object | `OptimResults` object | `sbiofit` | `sbiofitmixed`

**Introduced in R2014a**

## NLMEResults object

Results object containing estimation results from nonlinear mixed-effects modeling

### Description

The `NLMEResults` object contains estimation results from fitting a nonlinear mixed-effects model using `sbiofitmixed`.

### Method Summary

<code>boxplot(NLMEResults)</code>	Create box plot showing the variation of estimated SimBiology model parameters
<code>covariateModel(NLMEResults)</code>	Return a copy of the covariate model that was used for the nonlinear mixed-effects estimation using <code>sbiofitmixed</code>
<code>fitted(NLMEResults)</code>	Return the simulation results of a fitted nonlinear mixed-effects model
<code>plot(NLMEResults)</code>	Compare simulation results to the training data, creating a time-course subplot for each group
<code>plotActualVersusPredicted(NLMEResults)</code>	Compare predictions to actual data, creating a subplot for each response
<code>plotResiduals(NLMEResults)</code>	Plot the residuals for each response, using the time, group, or prediction as the x-axis
<code>plotResidualDistribution(NLMEResults)</code>	Plot the distribution of the residuals
<code>predict(NLMEResults)</code>	Simulate and evaluate fitted SimBiology model
<code>random(NLMEResults)</code>	Simulate a SimBiology model, adding variations by sampling the error model

## Property Summary

FixedEffects	Table of the estimated fixed effects and their standard errors.
RandomEffects	Table of the estimated random effects for each group.
IndividualParameterEstimates	Table of estimated parameter values, including fixed and random effects.
PopulationParameterEstimates	Table of estimated parameter values, including only fixed effects.
RandomEffectCovarianceMatrix	Table of the covariance matrix of the random effects.
stats	Struct of statistics returned by the <code>nlmefit</code> and <code>nlmefitsa</code> algorithm.
CovariateNames	Cell array of character vectors specifying covariate names.
EstimatedParameterNames	Cell array of character vectors specifying estimated parameter names.

<p>ErrorModelInfo</p>	<p>Table describing the error models and estimated error model parameters.</p> <p>The table has one row with three variables: <code>ErrorModel</code>, <code>a</code>, and <code>b</code>. The <code>ErrorModel</code> variable is categorical. The variables <code>a</code> and <code>b</code> can be <code>NaN</code> when they do not apply to a particular error model.</p> <p>There are four built-in error models. Each model defines the error using a standard mean-zero and unit-variance (Gaussian) variable <math>e</math>, the function value <math>f</math>, and one or two parameters <math>a</math> and <math>b</math>. In <code>SimBiology</code>, the function <math>f</math> represents simulation results from a <code>SimBiology</code> model.</p> <ul style="list-style-type: none"> <li>• 'constant': <math>y = f + ae</math></li> <li>• 'proportional': <math>y = f + b f e</math></li> <li>• 'combined': <math>y = f + (a+b f )e</math></li> <li>• 'exponential': <math>y = f * \exp(ae)</math></li> </ul>
<p>EstimationFunction</p>	<p>Name of the estimation function which must be either 'nlmefit' or 'nlmefitsa'.</p>
<p>LogLikelihood</p>	<p>Maximized loglikelihood for the fitted model.</p>
<p>AIC</p>	<p>Akaike Information Criterion (AIC), calculated as <math>AIC = 2 * (-\text{LogLikelihood} + P)</math>, where <math>P</math> is the number of parameters. For details, see <code>nlmefit</code>.</p>
<p>BIC</p>	<p>Bayes Information Criterion (BIC), calculated as <math>BIC = -2 * \text{LogLikelihood} + P * \log(N)</math>, where <math>N</math> is the number of observations or groups, and <math>P</math> is the number of parameters. For details, see <code>nlmefit</code>.</p>
<p>DFE</p>	<p>Degrees of freedom for error, calculated as <math>DFE = N - P</math>, where <math>N</math> is the number of observations and <math>P</math> is the number of parameters.</p>



**Note** If you are using the `nlmefitsa` method, `LogLikelihood`, `AIC`, and `BIC` properties are empty by default. To calculate these values, specify the `'LogLikMethod'` option of `nlmefitsa` when you run `sbiofitmixed` as follows.

```
opt.LogLikMethod = 'is';  
fitResults = sbiofitmixed(...,'nlmefitsa',opt);
```

---

## See Also

`nlmefit` | `nlmefitsa` | `sbiofit` | `sbiofitmixed`

**Introduced in R2014a**

## OptimResults object

Estimation results object, subclass of `LeastSquaresResults`

### Description

The `OptimResults` object contains estimation results from fitting a SimBiology model to data using the `sbiofit` function with any supported algorithm except `nlinfit`. See the `sbiofit` function for a list of supported algorithms.

### Method Summary

The `OptimResults` object shares all methods of the `LeastSquaresResults` object.

### Property Summary

The `OptimResults` object shares all properties of the `LeastSquaresResults` object and has the following additional properties.

<code>ExitFlag</code>	An exit flag specific to the estimation function.
<code>Output</code>	A struct of additional outputs specific to the estimation function.

---

**Note** See the reference page of the specific algorithm to get more information on the values of `ExitFlag` and the `Output` structure.

---

### See Also

`LeastSquaresResults` object | `NLINResults` object | `sbiofit` | `sbiofitmixed`

**Introduced in R2014a**

# Parameter object

Parameter and scope information

## Description

The parameter object represents a *parameter*, which is a quantity that can change or can be constant. SimBiology parameters are generally used to define rate constants. You can add parameter objects to a model object or a kinetic law object. The scope of a parameter depends on where you add the parameter object: If you add the parameter object to a model object, the parameter is available to all reactions in the model and the `Parent` property of the parameter object is `SimBiology.Model`. If you add the parameter object to a kinetic law object, the parameter is available only to the reaction for which you are using the kinetic law object and the `Parent` property of the parameter object is `SimBiology.KineticLaw`.

See “Property Summary” on page 2-272 for links to parameter object property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the graphical user interface.

## Constructor Summary

<code>addparameter (model, kineticlaw)</code>	Create parameter object and add to model or kinetic law object
---	--

## Method Summary

copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
findUsages (species, parameter, compartment)	Find out how a species, parameter, or compartment is used in a model
get (any object)	Get object properties
rename (compartment, parameter, species, reaction)	Rename object and update expressions
set (any object)	Set object properties

## Property Summary

ConstantValue	Specify variable or constant parameter value
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object
Value	Assign value to parameter object
ValueUnits	Parameter value units

## See Also

AbstractKineticLaw object, Configset object, KineticLaw object, Model object, Reaction object, Root object, Rule object, Species object

**Introduced in R2006b**

# PKCompartment object

Used by PKModelDesign to create SimBiology model

## Description

The PKCompartment object is used by the PKModelDesign object to construct a SimBiology model for pharmacokinetic modeling. PKCompartment holds the following information:

- Name of the compartment
- Dosing type
- Elimination type
- Whether the drug concentration in this compartment is reported

The PKCompartment class is a subclass of the hgsetget class which is a subclass of the handle class. For more information on the inherited methods, see hgsetget, and handle.

## Construction

addCompartment (PKModelDesign)      Add compartment to PKModelDesign object

## Method Summary

get (any object)	Get object properties
set (any object)	Set object properties

## Property Summary

DosingType	Drug dosing type in compartment
EliminationType	Drug elimination type from compartment
HasLag	Lag associated with dose targeting compartment
HasResponseVariable	Compartment drug concentration reported
Name	Specify name of object

## See Also

“Create Pharmacokinetic Models” in the SimBiology User's Guide, PKModelDesign object

**Introduced in R2009a**

## PKData object

Define roles of data set columns

---

**Note** PKData object will be removed in a future release. Use groupedData object instead.

---

### Description

The properties of the PKData object specify what each column in the data represents. The PKData object specifies which columns in the data set represent the following:

- The grouping variable
- The independent and dependent variables
- The dose
- The rate (only if infusion is the dosing type)
- The covariates

This information is used by the fitting functions, `sbionlmeFit` and `sbionlinfit`.

To create the PKData object specify:

```
pkDataObject = PKData(data);
```

Where `data` is the imported data set.

The PKData class is a subclass of the `hgsetget` class, which is a subclass of the `handle` class. For more information on the inherited methods, see `hgsetget` and `handle`.

### Construction

PKData

Create PKData object

## Method Summary

<code>get</code> (any object)	Get object properties
<code>getCovariateData</code> (pkdata)	Create design matrix needed for fit
<code>set</code> (any object)	Set object properties

## Property Summary

<code>CovariateLabels</code>	Identify covariate columns in data set
<code>DataSet</code>	Dataset object containing imported data
<code>DependentVarLabel</code>	Identify dependent variable column in data set
<code>DependentVarUnits</code>	Response units in PKData object
<code>DoseLabel</code>	Dose column in data set
<code>DoseUnits</code>	Dose units in PKData object
<code>GroupID</code>	Integer identifying each group in data set
<code>GroupLabel</code>	Identify group column in data set
<code>GroupNames</code>	Unique values from <code>GroupLabel</code> in data set
<code>IndependentVarLabel</code>	Identify independent variable column in data set
<code>IndependentVarUnits</code>	Time units in PKData object
<code>RateUnits</code>	Units for dose rate
<code>RateLabel</code>	Rate of infusion column in data set

## See Also

`PKModelDesign` object

**Introduced in R2009a**



# PKModelDesign object

Helper object to construct pharmacokinetic model

## Description

Use the `PKModelDesign` object to construct a `SimBiology` model for PK modeling. The `PKModelDesign` object lets you specify the number of compartments, the type of dosing, and method of elimination which you then use to construct the `SimBiology` model object with the necessary compartments, species, reactions, rules, and events.

```
pkm = PKModelDesign;
```

Use the `addCompartment` method to add a compartment with a specified dosing and elimination. `addCompartment` adds each subsequent compartment and connects it to the previous compartment using a reversible reaction. This reaction models the flux between compartments in a PK model.

The `construct` method uses the `PKModelDesign` object to create a `SimBiology` model object.

The `PKModelDesign` class is a subclass of the `hgsetget` class, which is a subclass of the `handle` class. For more information on the inherited methods see `hgsetget` and `handle`.

## Construction

`PKModelDesign`

Create `PKModelDesign` object

## Method Summary

addCompartment (PKModelDesign)	Add compartment to PKModelDesign object
construct (PKModelDesign)	Construct SimBiology model from PKModelDesign object
get (any object)	Get object properties
set (any object)	Set object properties

## Property Summary

PKCompartments	Hold compartments in PK model
----------------	-------------------------------

## See Also

“Create Pharmacokinetic Models” in the SimBiology User's Guide, PKCompartment object

**Introduced in R2009a**

## PKModelMap object

Define SimBiology model components' roles

---

**Note** PKModelMap object will be removed in a future release. Use a combination of estimatedInfo object, CovariateModel object, cell array of character vectors, and sbiodose. See sbiofit and sbiofitmixed for illustrated examples.

---

### Description

The PKModelMap object holds information about the dosing type, and defines which components of a SimBiology model represent the observed response, the dose, and the estimated parameters.

The PKModelMap class is a subclass of the hgsetget class which is a subclass of the handle class. For more information on the inherited methods see, hgsetget, and handle.

### Construction

PKModelMap

Create PKModelMap object

### Method Summary

get (any object)

Get object properties

set (any object)

Set object properties

## Property Summary

Dosed	Dosed object name
DosingType	Drug dosing type in compartment
Estimated	Names of parameters to estimate
LagParameter	Parameter specifying time lag for doses
Observed	Measured response object name
ZeroOrderDurationParameter	Zero-order dose absorption duration

## See Also

PKModelDesign object

**Introduced in R2009a**

## plot(LeastSquaresResults, OptimResults, NLINResults)

Compare simulation results to the training data, creating a time-course subplot for each group

### Syntax

```
plot(resultsObj)
```

### Description

plot(resultsObj) returns a figure displaying the comparison between simulation results to the training data, with a time-course subplot for each group.

### Input Arguments

#### resultsObj — Estimation results

OptimResults object | NLINResults object | vector of results objects

Estimation results, specified as an OptimResults object or NLINResults object, or vector of results objects which contains estimation results from running sbiofit.

### See Also

NLINResults object | OptimResults object | sbiofit

**Introduced in R2014a**

## plot(NLMEResults)

Compare simulation results to the training data, creating a time-course subplot for each group

### Syntax

```
plot(resultsObj)
plot(resultsObj, 'ParameterType', value)
```

### Description

`plot(resultsObj)` compares simulation results to the training data, creating a time-course subplot for each group.

`plot(resultsObj, 'ParameterType', value)` uses the individual or population parameter estimates as specified by `value`. The two choices for `value` are 'population' or 'individual' (default).

### Input Arguments

#### **resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results returned by `sbiofitmixed`.

#### **value** — Parameter type

character vector | string

Parameter type, specified as a character vector or string which must be one of the following: 'individual' (default) or 'population'.

## See Also

NLMEResults object | sbiofitmixed

**Introduced in R2014a**

## plot

**Package:** SimBiology.fit

Plot parameter confidence interval results

## Syntax

```
fh = plot(paraCI)
fh = plot(paraCI,Name,Value)
```

## Description

`fh = plot(paraCI)` plots confidence intervals from `paraCI`, a `ParameterConfidenceInterval` object or vector of objects.

- If the estimation status of a confidence interval (`paraCI.Results.Status` on page 2-0 ) is `success`, the `plot` function uses the first default color (blue) to plot a line and a centered dot for every parameter estimate. The function also plots a box to indicate the confidence intervals.
- If the status is `constrained` or `estimable`, the function uses the second default color (red) and plots a line, centered dot, and box to indicate the confidence intervals.
- If the status is `not estimable`, the function plots only a line and a centered cross in red.
- If there are any transformed parameters with estimated values that are 0 (for the `log` transform) and 0 or 1 (for the `probit` or `logit` transform), no confidence intervals are plotted for those parameter estimates.

`fh = plot(paraCI,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Examples

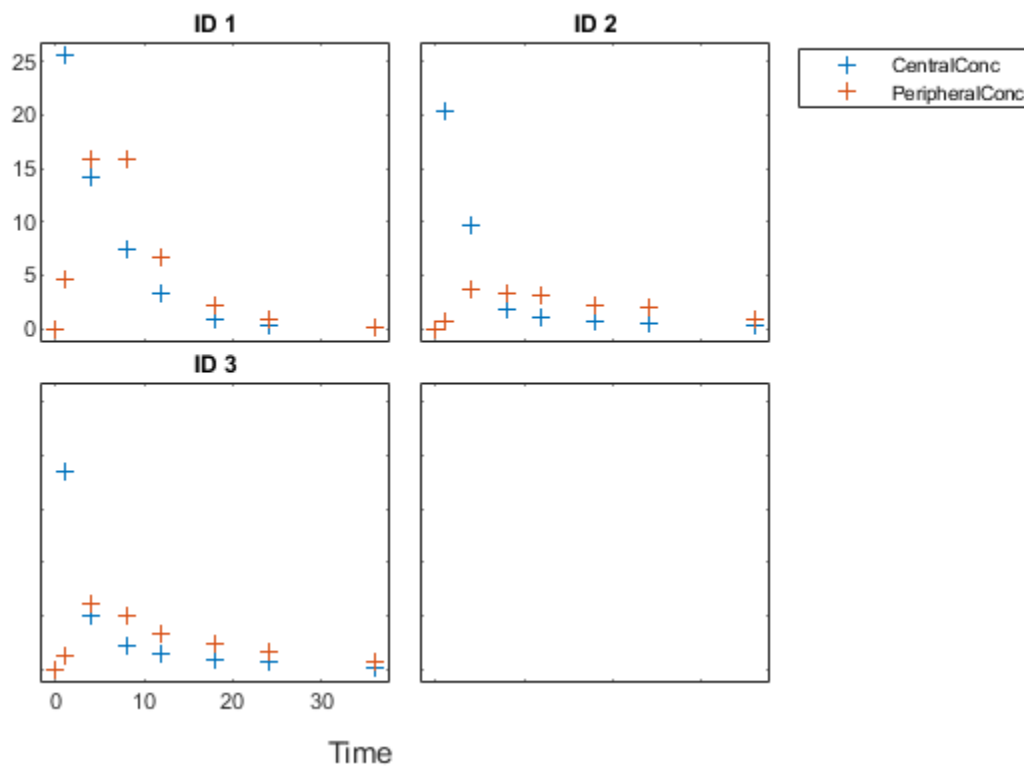


## Plot Confidence Intervals of Estimated PK Parameters

### Load Data

Load the sample data to fit.

```
clear all
load data10_32R.mat
gData = groupedData(data);
gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};
sbiotrellis(gData,'ID','Time',{'CentralConc','PeripheralConc'},'Marker','+',...
            'LineStyle','none');
```



### **Create Model**

Create a two-compartment model.

```
pkmd          = PKModelDesign;
pkc1          = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2          = addCompartment(pkmd, 'Peripheral');
model         = construct(pkmd);
configset     = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

### **Define Dosing**

Define the infusion dose.

```
dose          = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime = 0;
dose.Amount    = 100;
dose.Rate      = 50;
dose.AmountUnits = 'milligram';
dose.TimeUnits  = 'hour';
dose.RateUnits  = 'milligram/hour';
```

### **Define Parameters**

Define parameters to estimate.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam = estimatedInfo(paramsToEstimate, ...
                               'InitialValue', [1 1 1 1], ...
                               'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

### **Fit Model**

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

Perform a pooled fit, that is, one set of estimated parameters for all patients.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true);
```

## Compute Confidence Intervals for Estimated Parameters

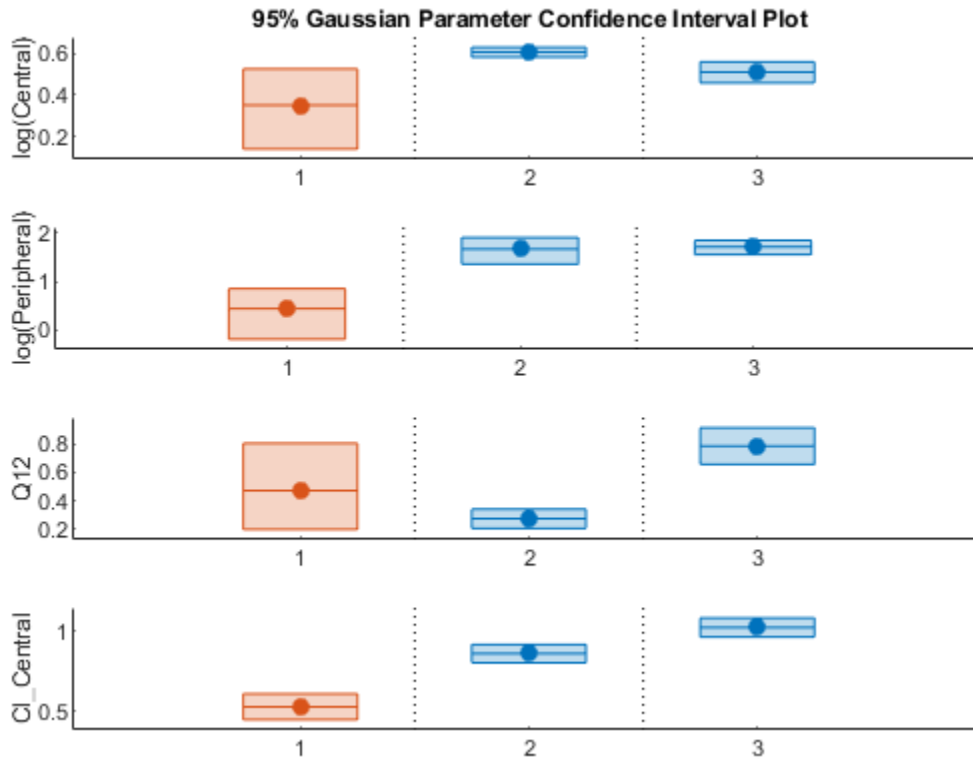
Compute 95% confidence intervals for each estimated parameter in the unpooled fit using the Gaussian approximation.

```
ciParamUnpooled = sbioparameterci(unpooledFit);
```

Plot the confidence intervals. If the estimation status of a confidence interval is `success`, it is plotted in blue (the first default color). Otherwise, it is plotted in red (the second default color), which indicates that further investigation into the fitted parameters might be required. If the confidence interval is `not estimable`, then the function plots a red line and centered cross. If there are any transformed parameters with estimated values that are 0 (for the log transform) and 1 or 0 (for the probit or logit transform), then no confidence intervals are plotted for those parameter estimates. To see the color order, type `get(groot, 'defaultAxesColorOrder')`.

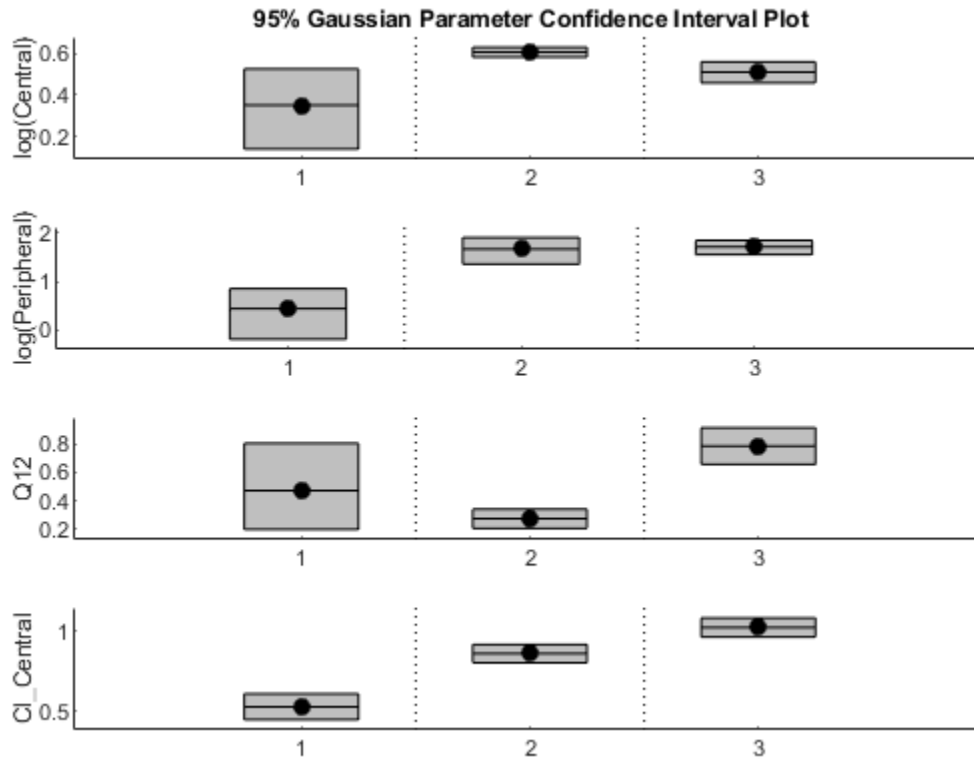
Groups are displayed from left to right in the same order that they appear in the `GroupNames` property of the object, which is used to label x-axis. The y-labels are the transformed parameter names.

```
plot(ciParamUnpooled)
```



Plot using a single color.

```
plot(ciParamUnpooled, 'Color', [0 0 0])
```

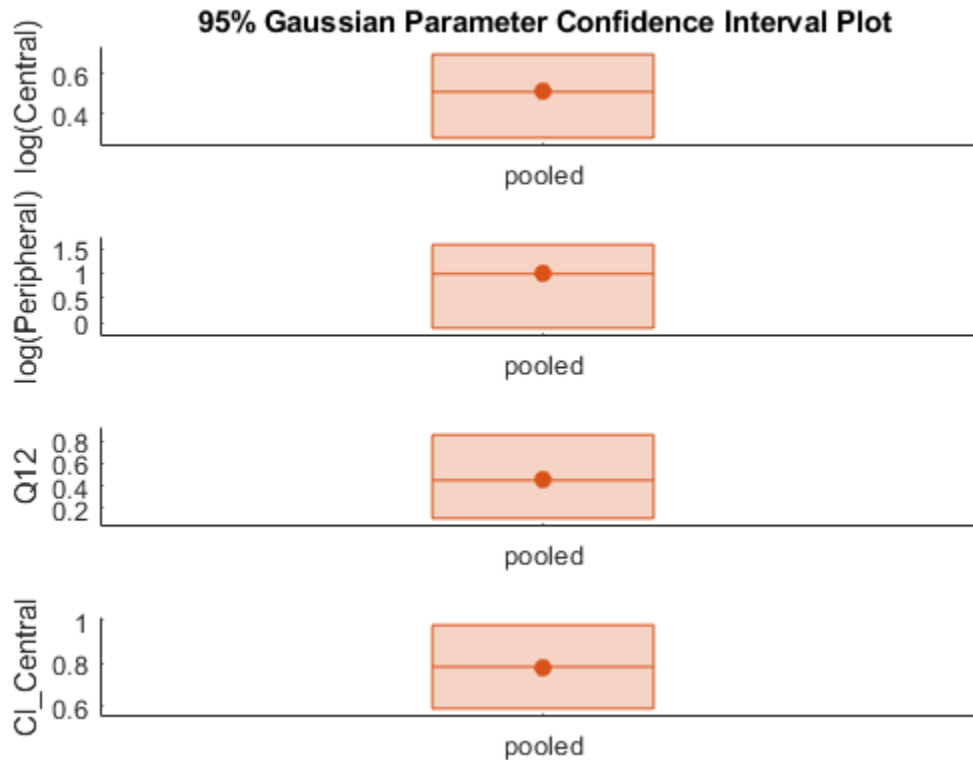


Compute the confidence intervals for the pooled fit.

```
ciParamPooled = sbioparameterci(pooledFit);
```

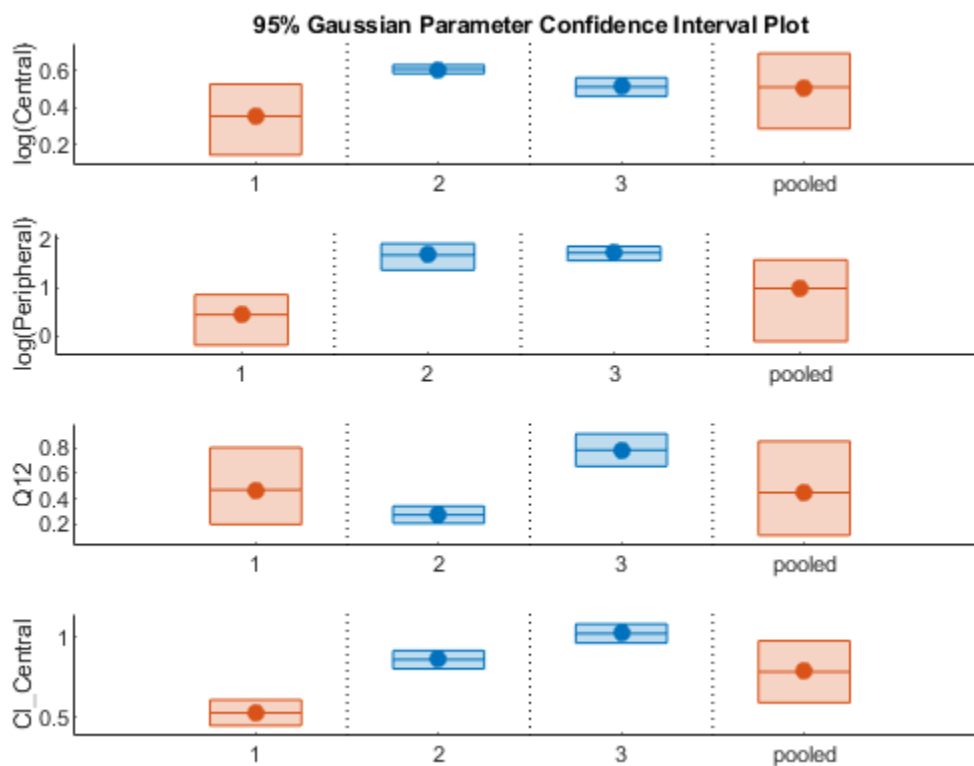
Plot the confidence intervals. The group name is labeled as "pooled" to indicate such fit.

```
plot(ciParamPooled)
```



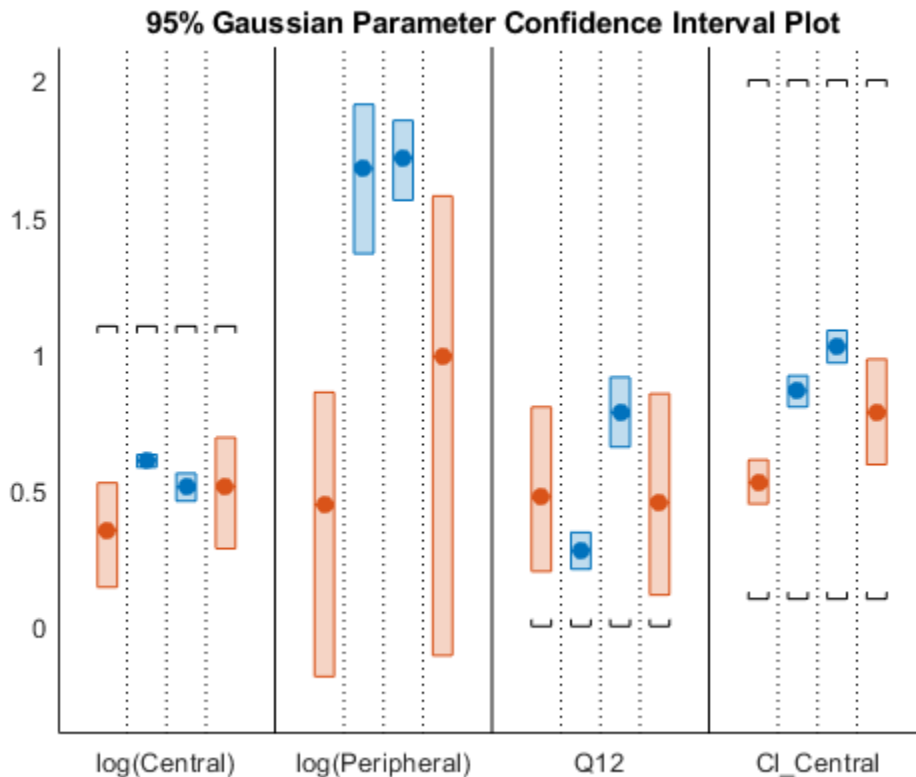
Plot all the confidence interval results together. By default, the confidence interval for each parameter estimate is plotted in a separate axes. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets (if visible in the parameter range being plotted).

```
ciAll = [ciParamUnpooled;ciParamPooled];  
plot(ciAll)
```



You can also plot all confidence intervals on one axes grouped by parameter estimates using the 'Grouped' layout.

```
plot(ciAll, 'Layout', 'Grouped')
```



In this layout, you can point to the center marker of each confidence interval to see the group name. Each estimated parameter is separated by a vertical black line. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets (if visible in the parameter range being plotted). Note the different scales on the y-axis due to parameter transformations. For instance, the y-axis of Q12 is in the linear scale, but that of Central is in the log scale due to its log transform.

### Compute Confidence Intervals Using Profile Likelihood

Compute 95% confidence intervals for each estimated parameter in the unpooled fit using the profile likelihood approach.

```
ciParamUnpooledProf = sbioparameterci(unpooledFit, 'Type', 'profilelikelihood');
```

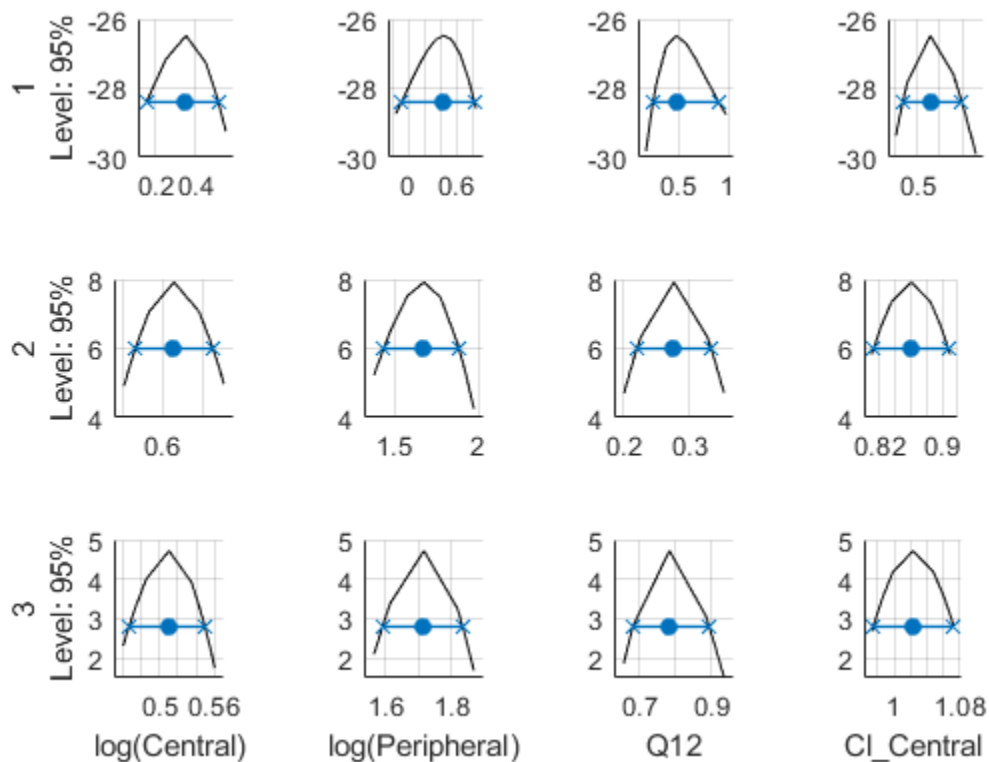


Compute the confidence intervals for the pooled fit.

```
ciParamPooledProf = sbioparameterci(pooledFit,'Type','profilelikelihood');
```

Plot the profile likelihood curves for the unpooled fit. The parameter bounds defined in the original fit are displayed by vertical dotted lines (if visible in the parameter range being plotted). The confidence interval is indicated by two crosses and a line in between them. The center dot denotes the parameter estimate. The profile likelihood is always plotted in the log scale. The x-axis scale depends on whether the parameter is transformed (log, probit, or logit scale) or not (linear scale).

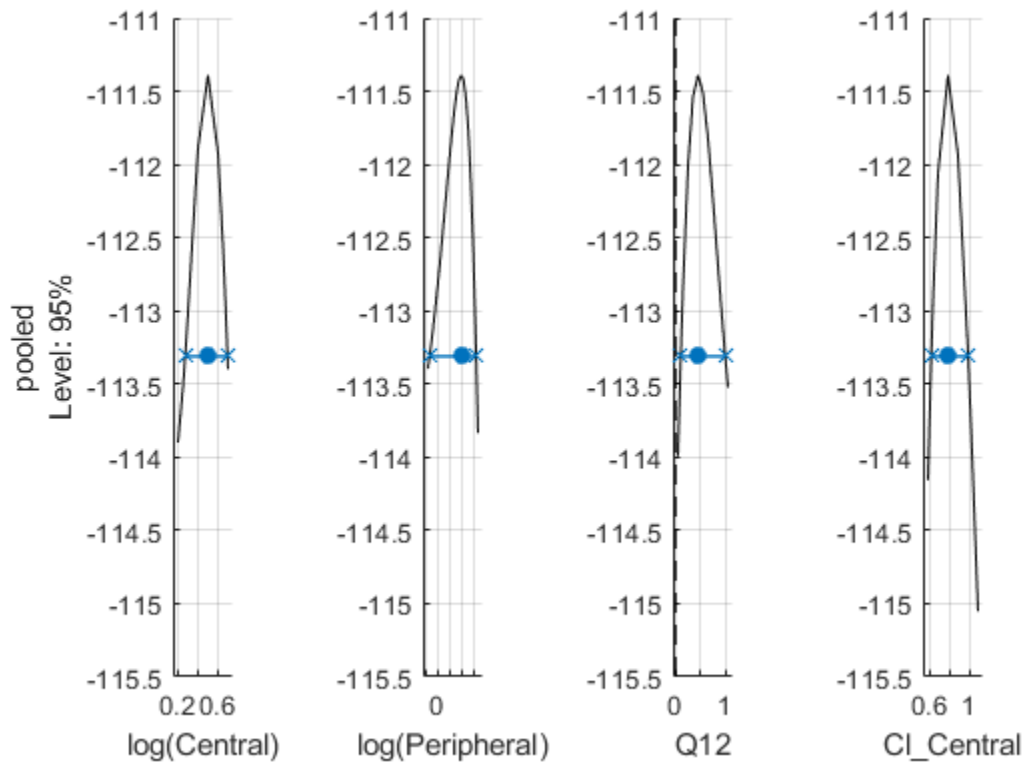
```
plot(ciParamUnpooledProf,'ProfileLikelihood',true);
```



Each group is plotted in a separate row, and each parameter is plotted in a separate column.

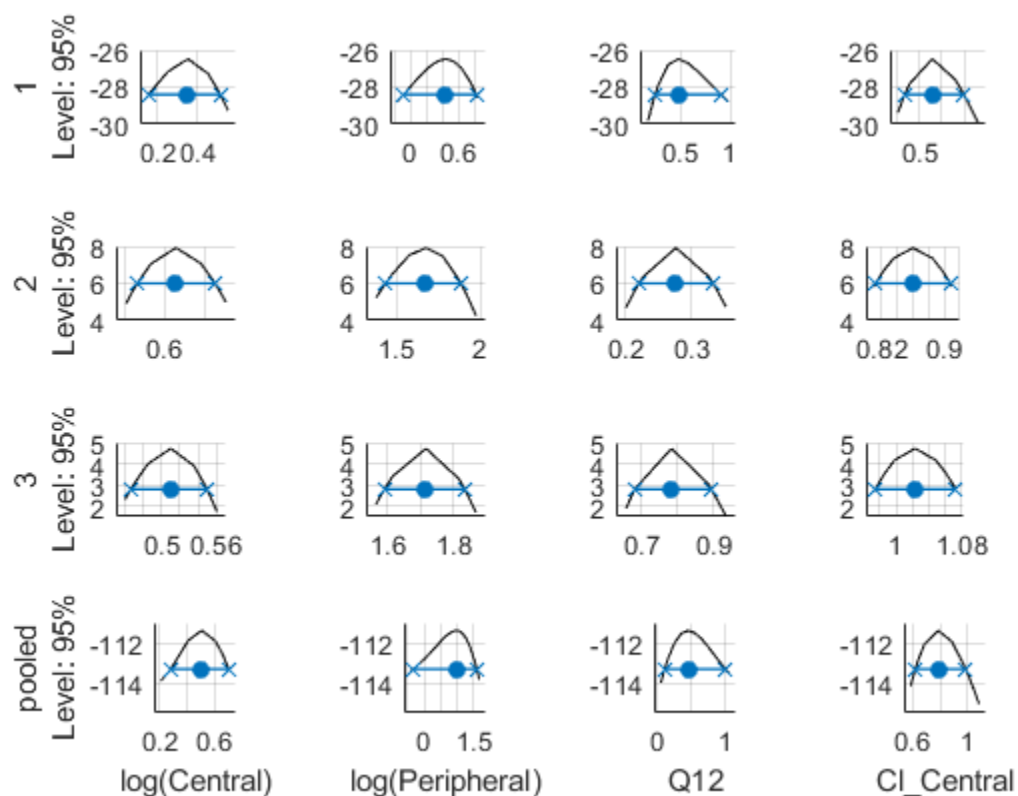
Plot the curves for the pooled fit.

```
plot(ciParamPooledProf, 'ProfileLikelihood', true);
```



Plot all the confidence interval results together in the same figure.

```
plot([ciParamUnpooledProf;ciParamPooledProf], 'ProfileLikelihood', true);
```



## Input Arguments

### paraCI — Parameter confidence interval results

ParameterConfidenceInterval object | vector

Parameter confidence interval results, specified as a ParameterConfidenceInterval object or a vector of objects.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes.

You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'ProfileLikelihood', true` specifies to plot the profile likelihood curves.

### **Color — Red-Green-Blue color triplet**

three-element row vector

Red-Green-Blue color triplet, specified as the comma-separated pair consisting of `'Color'` and a three-element row vector. By default, confidence intervals that are not limited by the parameter bounds specified in the original fit are plotted using the first default color (blue), and those that are limited by the bounds are plotted using the second default color (red). If the confidence interval is not estimable, it is also plotted in red. To see the default color order, enter `get(groot, 'defaultAxesColorOrder')` or see the `"ColorOrder"` (MATLAB) property.

---

**Tip** Use this name-value pair when you want to create plots with a single color, for instance, for publication purposes.

---

Example: `'Color', [0 0 0]`

### **ProfileLikelihood — Logical scalar to display profile likelihood curves**

false (default) | true

Logical scalar to display profile likelihood curves for the `profileLikelihood` confidence intervals, specified as the comma-separated pair consisting of `'ProfileLikelihood'` and true or false.

The confidence interval is indicated by two crosses with a line in between them. A center dot denotes the parameter estimate. The `plot` function uses the first default color (blue) for successfully computed confidence intervals. Otherwise, the function uses the second default color (red). A vertical dotted line marks the parameter bounds defined in the original fit.

If there are multiple groups, each group is plotted in a separate row and each parameter is plotted in a separate column. The labels for the x-axis are the transformed parameter names (the `TransformedName` property of the `estimatedInfo` object used in the original fit). The labels for the y-axis are the group names (the `GroupNames` property of the confidence interval object) and the confidence level.

The profile likelihood curve is always plotted in the log scale. The x-axis scale depends on whether the parameter is transformed (log, probit, or logit scale) or not (linear scale).

Example: `'ProfileLikelihood', true`

### **Layout — Axes layout to display parameter confidence intervals**

`'split'` (default) | `'grouped'`

Axes layout to display parameter confidence intervals, specified as the comma-separated pair consisting of `'Layout'` and a character vector `'split'` (default) or `'grouped'`.

The `'split'` layout displays the confidence interval for each parameter estimate on a separate axes.

The `'grouped'` layout displays all confidence intervals on one axes grouped by parameter estimates. Each estimated parameter is separated by a vertical black line.

In both cases, the parameter bounds defined in the original fit are marked by square brackets. The function uses vertical dotted lines to group confidence intervals of parameter estimates that have been computed in a common fit.

Example: `'Layout', 'grouped'`

## **Output Arguments**

### **fh — Figure handle**

handle

Figure handle of the plot, returned as a figure handle.

## **See Also**

`ParameterConfidenceInterval` | `sbioparameterci`

**Introduced in R2017b**

# plot

**Package:** SimBiology.fit

Plot confidence interval results for model predictions

## Syntax

```
fh = plot(predCI)
```

## Description

`fh = plot(predCI)` plots confidence interval results from `predCI`, a `PredictionConfidenceInterval` object or vector of objects.

The function plots the observation data points as black plus signs and the model predictions as solid lines.

- If the status of confidence interval (`predCI.Status` on page 2-0 ) is `constrained` or `not estimable`, the function uses the second default color (red) to plot the confidence intervals.
- Otherwise, the function uses the first default color (blue) and plots the confidence intervals as shaded areas.

## Examples

### Plot Confidence Intervals of PK Model Predictions

#### Load Data

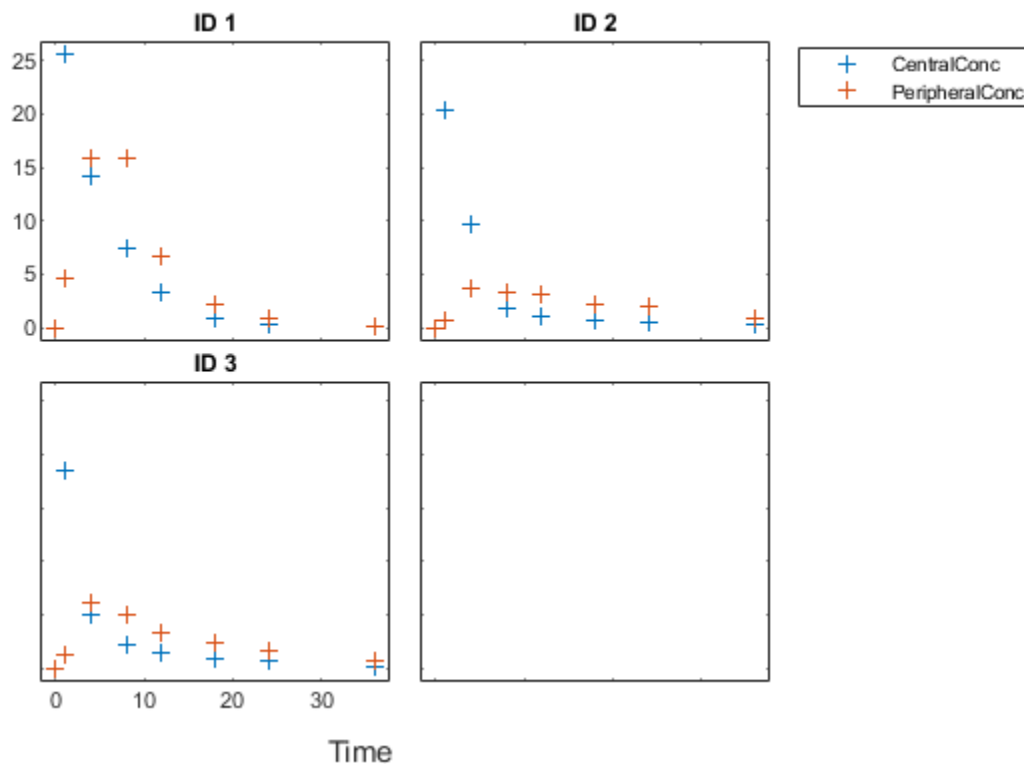
Load the sample data to fit.

```
clear all
load data10_32R.mat
gData = groupedData(data);
```

```

gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};
sbiotrellis(gData,'ID','Time',{ 'CentralConc','PeripheralConc'}, 'Marker','+',...
            'LineStyle','none');

```



## Create Model

Create a two-compartment model.

```

pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd,'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2 = addCompartment(pkmd,'Peripheral');

```

```
model          = construct(pkmd);
configset      = getconfigset(model);
configset.CompileOptions.UnitConversion = true;
```

### Define Dosing

Define the infusion dose.

```
dose           = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime = 0;
dose.Amount    = 100;
dose.Rate      = 50;
dose.AmountUnits = 'milligram';
dose.TimeUnits  = 'hour';
dose.RateUnits  = 'milligram/hour';
```

### Define Parameters

Define parameters to estimate.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam  = estimatedInfo(paramsToEstimate, ...
                                'InitialValue', [1 1 1 1], ...
                                'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

### Fit Model

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

### Compute Confidence Intervals for Model Predictions

Compute 95% confidence intervals for predicted model responses in the unpooled fit using the Gaussian approximation.

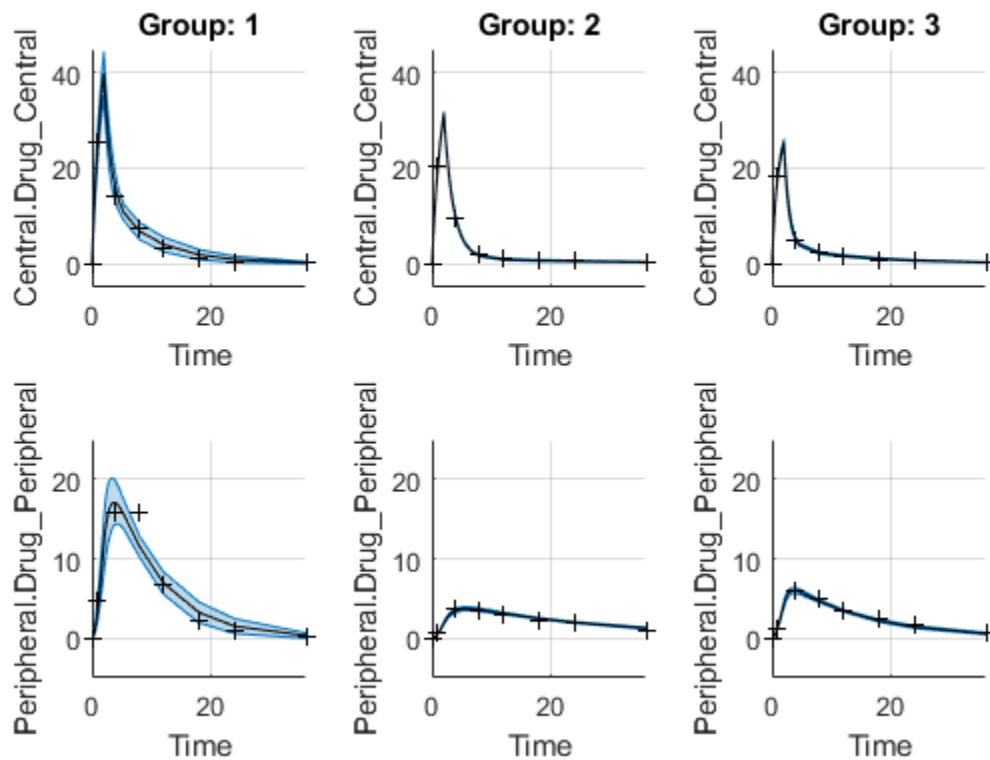
```
ciPredUnpooled = sbiopredictionci(unpooledFit);
```

Plot the confidence intervals. If the estimation status of a confidence interval is `constrained` or `not estimable`, the function uses the second default color (red). Otherwise, the function uses the first default color (blue). To see the color order, type `get(groot, 'defaultAxesColorOrder')`.



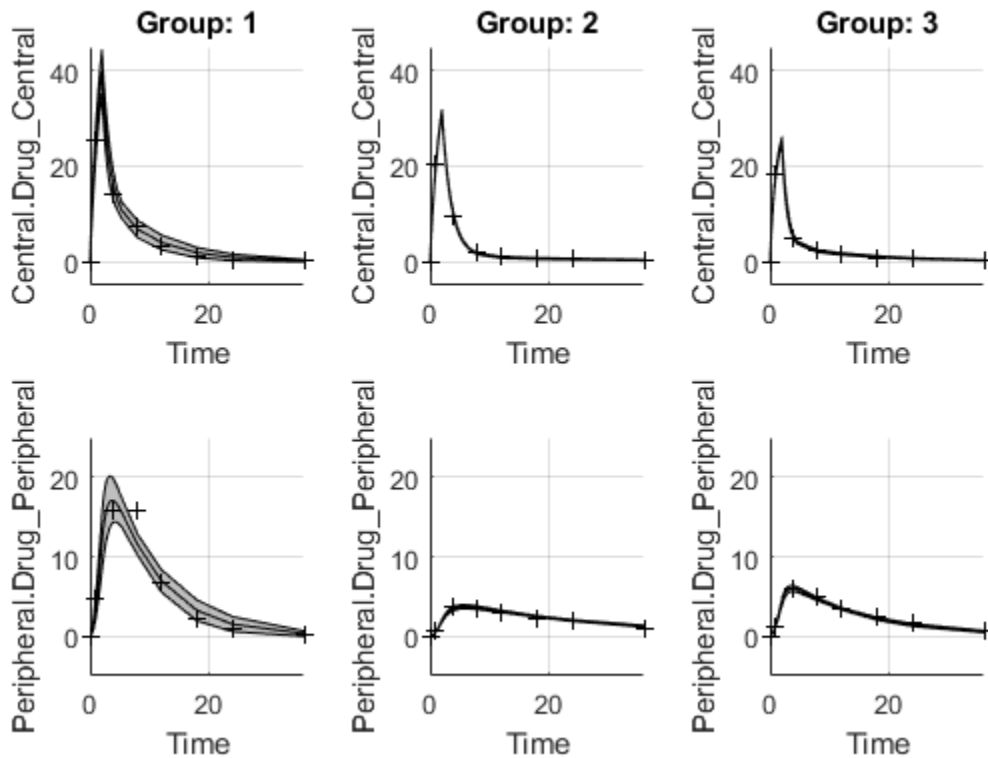
Each group is displayed in each column, from left to right, in the same order that they appear in the GroupNames property of the object, which is used to label each column.

```
plot(ciPredUnpooled)
```



Plot using a single color (black).

```
plot(ciPredUnpooled, 'Color', [0 0 0])
```



## Input Arguments

### **predCI — Confidence interval results for model predictions**

PredictionConfidenceInterval object | vector

Confidence interval results for model predictions, specified as a PredictionConfidenceInterval object or a vector of objects.

If there are multiple groups, each group is displayed in each column, from left to right, in the same order that they appear in the GroupNames property of the object. Each row represents a model response.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'Color', [0 0 0]` specifies to plot using only the color black.

### **Color** — Red-Green-Blue color triplet

three-element row vector

Red-Green-Blue color triplet, specified as the comma-separated pair consisting of `'Color'` and a three-element row vector. By default, confidence intervals that are not limited by parameter bounds specified in the original fit are plotted using the first default color (blue), and those that are limited by the bounds are plotted using the second default color (red). If the confidence interval is not estimable, it is also plotted in red. To see the default color order, enter `get(groot, 'defaultAxesColorOrder')` or see the `"ColorOrder"` (MATLAB) property.

---

**Tip** Use this name-value pair when you want to create plots with a single color, for instance, for publication purposes.

---

Example: `'Color', [0 0 0]`

## Output Arguments

### **fh** — Figure handle

handle

Figure handle of the plot, returned as a figure handle.

## See Also

`ParameterConfidenceInterval` | `PredictionConfidenceInterval` | `sbioparameterci` | `sbiopredictionci`

**Introduced in R2017b**

# plotActualVersusPredicted(LeastSquaresResults,OptimResults,NLINResults)

Compare predictions to actual data, creating a subplot for each response

## Syntax

```
plotActualVersusPredicted(resultsObj)
```

## Description

`plotActualVersusPredicted(resultsObj)` returns a figure displaying the comparison between predictions to the actual data, with a subplot for each response.

## Input Arguments

### **resultsObj** — Estimation results

`OptimResults` object | `NLINResults` object | vector of results objects

Estimation results, specified as an `OptimResults` object or `NLINResults` object, or vector of results objects which contains estimation results from running `sbiofit`.

## See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## plotActualVersusPredicted(NLMEResults)

Compare predictions to actual data, creating a subplot for each response

### Syntax

```
plotActualVersusPredicted(resultsObj)
```

### Description

`plotActualVersusPredicted(resultsObj)` returns a figure displaying the comparison between predictions to the actual data, with a subplot for each response.

### Input Arguments

#### **resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results returned by `sbiofitmixed`.

### See Also

NLMEResults object | `sbiofitmixed`

**Introduced in R2014a**

# plotResiduals(LeastSquaresResults, OptimResults, NLINResults)

Plot residuals for each response, using time, group, or prediction as x-axis

## Syntax

```
plotResiduals(resultsObj, type)
```

## Description

`plotResiduals(resultsObj, type)` plots the residuals for each response, using the time, group, or predictions as the x-axis as specified by the argument `type`.

## Input Arguments

### **resultsObj** – Estimation results

`OptimResults` object | `NLINResults` object | vector of results objects

Estimation results, specified as an `OptimResults` object, `NLINResults` object, or a vector of results object which contains estimation results returned by `sbiofit`.

### **type** – x-axis option for residual plot

character vector | string

X-axis option for the residual plot, specified as a character vector or string that must be one of the following: 'Time', 'Group', or 'Predictions'.

## See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## plotResiduals(NLMEResults)

Plot the residuals for each response, using the time, group, or prediction as the x-axis

### Syntax

```
plotResiduals(resultsObj, type)
```

### Description

`plotResiduals(resultsObj, type)` plots the residuals for each response, using the time, group, or predictions as the x-axis as specified by the argument `type`.

### Input Arguments

#### **resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results returned by `sbiofitmixed`.

#### **type** — X-axis option for residual plot

character vector | string

X-axis option for the residual plot, specified as a character vector or string which must be one of the following: 'Time', 'Group', or 'Predictions'.

### See Also

NLMEResults object | sbiofitmixed

**Introduced in R2014a**



# plotResidualDistribution(LeastSquaresResults,OptimResults,NLINResults)

Plot the distribution of the residuals

## Syntax

```
plotResidualDistribution(resultsObj)
```

## Description

`plotResidualDistribution(resultsObj)` plots the distribution of the residuals.

## Input Arguments

### **resultsObj** — Estimation results

`OptimResults` object | `NLINResults` object | vector of results objects

Estimation results, specified as an `OptimResults` object or `NLINResults` object, or vector of results objects which contains estimation results from running `sbiofit`.

## See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## plotResidualDistribution(NLMEResults)

Plot the distribution of the residuals

### Syntax

```
plotResidualDistribution(resultsObj)
```

### Description

`plotResidualDistribution(resultsObj)` plots the distribution of the residuals.

### Input Arguments

#### **resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an NLMEResults object, which contains estimation results from running `sbiofitmixed`.

### See Also

NLMEResults object | `sbiofitmixed`

**Introduced in R2014a**

# predict(LeastSquaresResults,OptimResults,NLINResults)

Simulate and evaluate fitted SimBiology model

## Syntax

```
[ynew, parameterEstimates]= predict(resultsObj)  
[ynew, parameterEstimates]= predict(resultsObj, data, dosing)
```

## Description

[ynew, parameterEstimates]= predict(resultsObj) returns simulation results ynew and parameter estimates parameterEstimates of a fitted SimBiology model.

[ynew, parameterEstimates]= predict(resultsObj, data, dosing) returns simulation results ynew and estimated parameter values parameterEstimates from evaluating the fitted SimBiology model using the specified data and dosing information.

During simulations, predict uses the parameter values in the resultsObj.ParameterEstimates property. Use this method when you want to evaluate the fitted model and predict responses using new data and/or dosing information.

## Input Arguments

### resultsObj — Estimation results

OptimResults object | NLINResults object

Estimation results, specified as an OptimResults object or NLINResults object, which contains estimation results returned by sbiofit. It must be a scalar object.

### data — Output times or grouped data

vector | cell array of vectors | groupedData object

Output times or grouped data, specified as a vector, or cell array of vectors of output times, or `groupedData` object.

If it is a vector of time points, `predict` simulates the model with new time points using the parameter estimates from the results object `resultsObj`.

If it is a cell array of vectors of time points, `predict` simulates the model  $n$  times using the output times from each time vector, where  $n$  is the length of `data`.

If it is a `groupedData` object, it must have an independent variable such as `Time`. It must also have a group variable if the training data used for fitting has such variable. You can use a `groupedData` object to query different combinations of categories if the `resultsObj` contains parameter estimates for each category. `predict` simulates the model for each group with the specified categories. For instance, suppose you have a set of parameter estimates for sex category (males versus females), and age category (young versus old) in your training data. You can use `predict` to simulate the responses of an old male (or any other combination) although such patient may not exist in the training data.

If the `resultsObj` is from estimating category-specific parameters, `data` must be a `groupedData` object.

---

**Note** If `UnitConversion` is turned on for the underlying SimBiology model that was used for fitting and `data` is a `groupedData` object, `data` must specify valid variable units via `data.Properties.VariableUnits` property. If it is a numeric vector or cell array of vectors of time points, `predict` uses the model's `TimeUnits`.

---

### **dosing — Dosing information**

[ ] | 2-D matrix of SimBiology dose objects

Dosing information, specified as an empty array [ ] or 2-D matrix of SimBiology dose objects (`ScheduleDose` object or `RepeatDose` object). If `dosing` is a matrix of dose objects, the matrix must contain default doses or be consistent with the original dosing data used with `sbiofit`. That is, dose objects in `dosing` must have the same values for dose properties (such as `TargetName`) or must be parameterized in the same way as the original dosing data. For instance, suppose that the original dosing matrix has two columns of doses, where the doses in the first column target species  $x$  and those in the second column target species  $y$ . Then `dosing` must have doses in the first column targeting species  $x$  and those in the second column targeting species  $y$ .

- If empty, no doses are applied during simulation, even if the model has active doses.
- If not empty, the matrix must have a single row or one row per group in the input data. If it has a single row, the same doses are applied to all groups during simulation. If it has multiple rows, each row is applied to a separate group, in the same order as the groups appear in the input data. If some groups (or time vectors) require more doses than others, then fill in the matrix with default (dummy) doses.
- Multiple columns are allowed so that you can apply multiple dose objects to each group or time vector. All doses in a column must be default doses or must reference the same components in the model (for instance, the doses must have the same dose target `TargetName`) and must have consistent parameterized properties as in the original dosing data used with `sbiofit`. For example, if the `Amount` property of a dose used in the original `sbiofit` call is parameterized to a model-scoped parameter `'A'`, all doses for the corresponding group (column) in `dosing` must have the `Amount` property parameterized to `'A'`.
- A default dose has default values for all properties, except for the `Name` property. Create a default dose as follows.

```
d1 = sbiodose('d1');
```

- In addition to manually constructing dose objects using `sbiodose`, if the input data is a `groupedData` object and has dosing information, you can use the `createDoses` method to construct doses from it.

The number of rows in the `dosing` matrix and the number of groups or output time vectors in `data` determine the total number of simulation results in the output `ynew`. For details, see the table in the `ynew` argument description.

---

**Note** If `UnitConversion` is turned on for the underlying SimBiology model that was used for fitting, `dosing` must specify valid amount and time units.

---

## Output Arguments

### **ynew** — Simulation results

vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects. The states reported in `ynew` are the states that were included in the `responseMap` input argument of `sbiofit` as well as any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the SimBiology model.

The total number of simulation results in `ynew` depends on the number of groups or output time vectors in `data` and the number of rows in the `dosing` matrix.

Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
1	0, that is, <code>dosing</code> is empty []	The total number of <code>SimData</code> objects in <code>ynew</code> is 1.  No doses are applied during simulation.
1	1	The total number of <code>SimData</code> objects in <code>ynew</code> is 1.  The given row of doses is applied during the simulation.
1	$N$	The total number of <code>SimData</code> objects in <code>ynew</code> is $N$ .  Each row of <code>dosing</code> is applied to each simulation.
$N$	0, that is, <code>dosing</code> is empty []	The total number of <code>SimData</code> objects in <code>ynew</code> is $N$ .  No doses are applied during simulation.
$N$	1	The total number of <code>SimData</code> objects in <code>ynew</code> is $N$ .  The same row of doses is applied to each simulation.

Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
$N$	$N$	The total number of SimData objects in ynew is $N$ .  Each row of dosing is applied to a separate group, in the same order that groups appear in data.
$M$	$N$	The function throws an error when $M \neq N$ .

### parameterEstimates – Estimated parameter values

table

Estimated parameter values, returned as a table. This is identical to resultsObj.ParameterEstimates property. The predict method uses these parameter values during simulation.

## Examples

### Estimate a Parameter from the Yeast G Protein Model

This example uses the yeast heterotrimeric G protein model and experimental data reported by [1]. For details about the model, see the **Background** section in “Parameter Scanning, Parameter Estimation, and Sensitivity Analysis in the Yeast Heterotrimeric G Protein Cycle”.

Load the G protein model.

```
sbioloadproject gprotein
```

Enter the experimental data containing the time course for the fraction of active G protein, as reported in the reference paper [1].

```
time = [0;10;30;60;110;210;300;450;600];
GaFracExpt = [0;0.35;0.4;0.36;0.39;0.33;0.24;0.17;0.2];
```

Create a `groupedData` object based on the experimental data.

```
tbl = table(time, GaFracExpt);  
grpData = groupedData(tbl);
```

Map the appropriate model component to the experimental data. In other words, indicate which species in the model corresponds to which response variable in the data. In this example, map the model parameter `GaFrac` to the experimental data variable `GaFracExpt` from `grpData`.

```
responseMap = 'GaFrac = GaFracExpt';
```

Use an `estimatedInfo` object to define the model parameter `kGd` as a parameter to be estimated.

```
estimatedParam = estimatedInfo('kGd');
```

Perform the parameter estimation.

```
fitResult = sbiofit(m1, grpData, responseMap, estimatedParam);
```

View the estimated parameter value of `kGd`.

```
fitResult.ParameterEstimates
```

```
ans =
```

Name	Estimate	StandardError
'kGd'	0.11	3.683e-05

Suppose you want to simulate the fitted model using different output times than those in the training data. You can use the `predict` method to do so.

Create a new variable `T` with different output times.

```
T = [0;10;50;80;100;150;300;350;400;450;500;550];
```

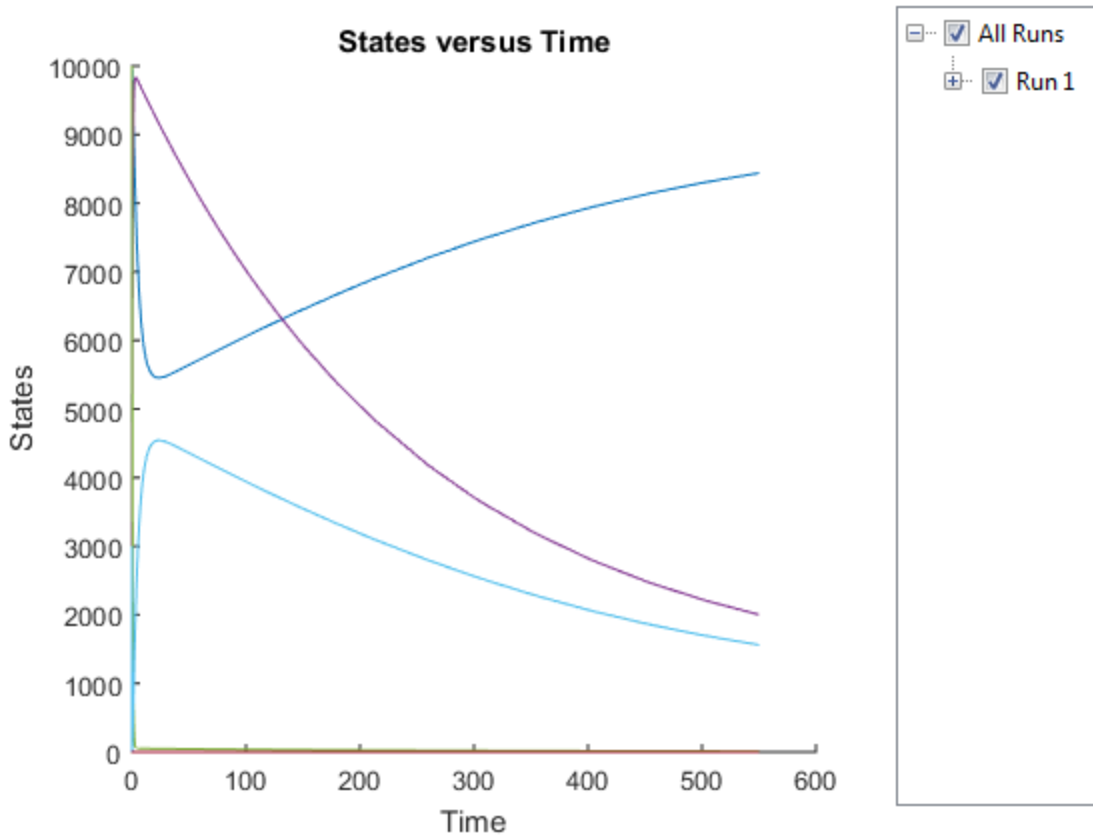
Use the `predict` method to simulate the fitted model on the new time points. No dosing was specified when you first ran `sbiofit`. Hence, you cannot use any dosing information with the `predict` method, and an empty array must be specified as the third input argument.

```
ynew = predict(fitResult, T, []);
```

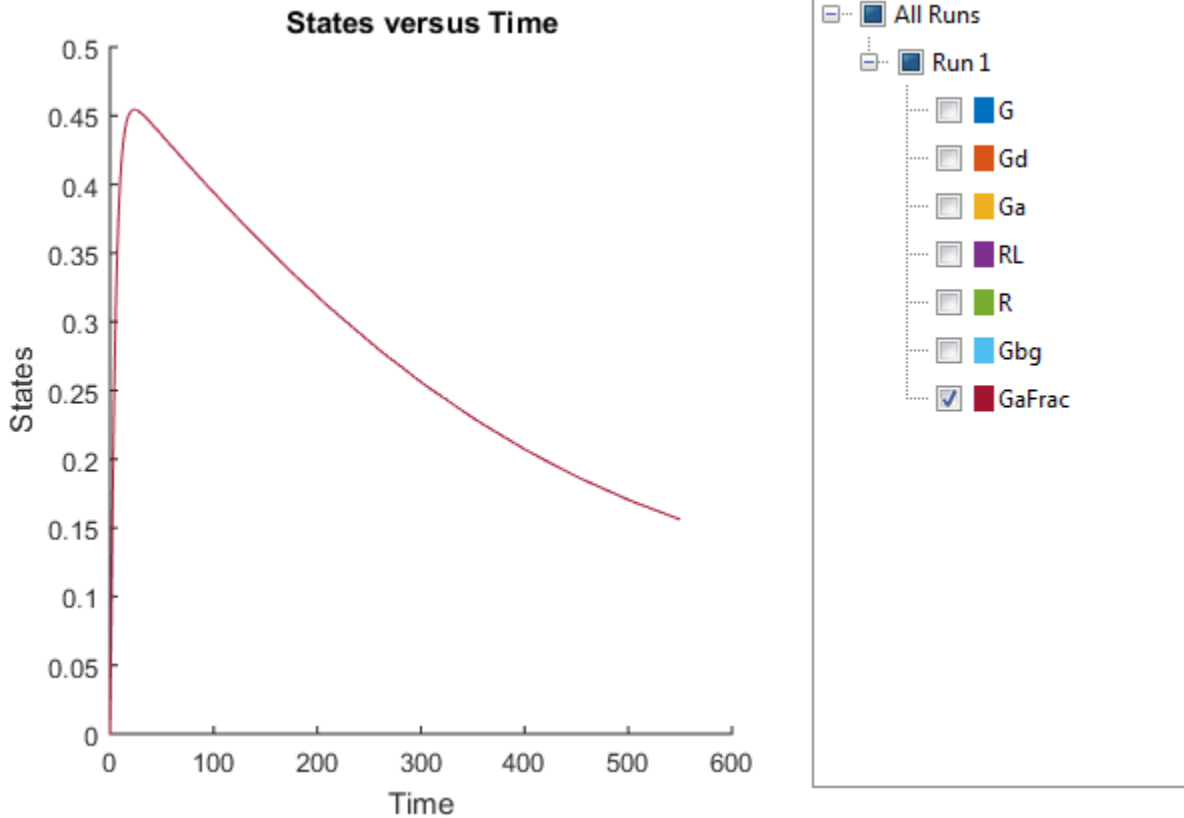


Plot the simulated data with the new output times.

```
sbioplot(ynew)
```



Expand **Run1** and only select **GaFrac** to display its simulated data.



### Estimate category-specific PK parameters

This example shows how to estimate category-specific (such as young versus old, male versus female) PK parameters using the profile data from multiple individuals using a two-compartment model. The parameters to estimate are the volumes of central and peripheral compartment, the clearance, and intercompartmental clearance.

The synthetic data used in this example contains the time course of plasma concentrations of multiple individuals after a bolus dose (100 mg) measured at different times for both central and peripheral compartments. It also contains categorical variables, namely Sex and Age.

```
clear
load(fullfile(matlabroot, 'examples', 'simbio', 'sd5_302RAgeSex.mat'));
```

Convert the data set to a `groupedData` object, which is the required data format for the fitting function `sbiofit`. A `groupedData` object allows you to set independent variable and group variable names (if they exist). Set the units of the `ID`, `Time`, `CentralConc`, `PeripheralConc`, `Age`, and `Sex` variables. The units are optional and only required for the `UnitConversion` feature, which automatically converts matching physical quantities to one consistent unit system.

```
gData = groupedData(data);
gData.Properties.VariableUnits = {'', 'hour', 'milligram/liter', 'milligram/liter', '', ''};
```

The `IndependentVariableName` and `GroupVariableName` properties have been automatically set to the `Time` and `ID` variables of the data.

```
gData.Properties
```

```
ans =
```

```

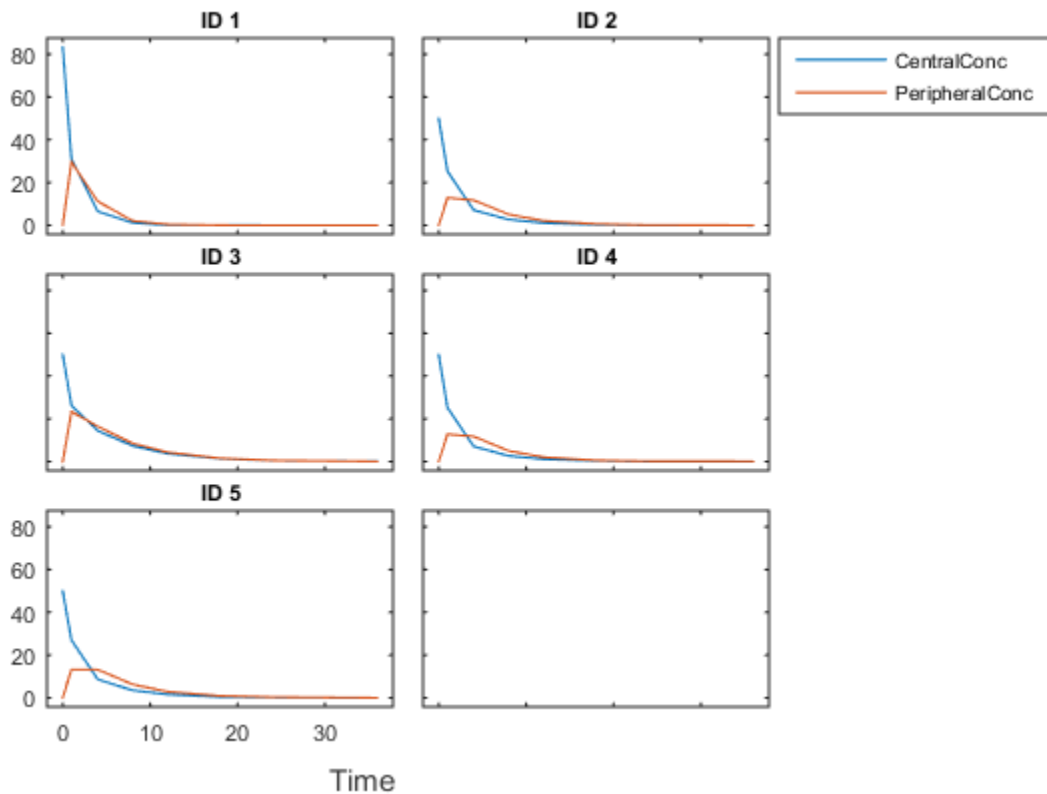
      Description: ''
VariableDescriptions: {}
      VariableUnits: {'' 'hour' 'milligram/liter' 'milligram/liter' '' ''}
DimensionNames: {'Row' 'Variable'}
      UserData: []
      RowNames: {}
      VariableNames: {'ID' 'Time' 'CentralConc' 'PeripheralConc' 'Sex' 'Age'}
GroupVariableName: 'ID'
IndependentVariableName: 'Time'
```

For illustration purposes, use the first five individual data for training and the 6th individual data for testing.

```
trainData = gData([gData.ID < 6],:);
testData = gData([gData.ID == 6],:);
```

Display the response data for each individual in the training set.

```
sbiotrellis(trainData, 'ID', 'Time', {'CentralConc', 'PeripheralConc'});
```



Use the built-in PK library to construct a two-compartment model with infusion dosing and first-order elimination where the elimination rate depends on the clearance and volume of the central compartment. Use the `configset` object to turn on unit conversion.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2 = addCompartment(pkmd, 'Peripheral');
model = construct(pkmd);
configset = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

Assume every individual receives a bolus dose of 100 mg at time = 0.

```
dose                = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime     = 0;
dose.Amount        = 100;
dose.AmountUnits   = 'milligram';
dose.TimeUnits     = 'hour';
```

The data contains measured plasma concentration in the central and peripheral compartments. Map these variables to the appropriate model components, which are `Drug_Central` and `Drug_Peripheral`.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
```

Specify the volumes of central and peripheral compartments `Central` and `Peripheral`, intercompartmental clearance `Q12`, and clearance `Cl_Central` as parameters to estimate. The `estimatedInfo` object lets you optionally specify parameter transforms, initial values, and parameter bounds. Since both `Central` and `Peripheral` are constrained to be positive, specify a log-transform for each parameter.

```
paramsToEstimate    = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam      = estimatedInfo(paramsToEstimate, 'InitialValue', [1 1 1 1]);
```

Use the `'CategoryVariableName'` property of the `estimatedInfo` object to specify which category to use during fitting. Use `'Sex'` as the group to fit for the clearance `Cl_Central` and `Q12` parameters. Use `'Age'` as the group to fit for the `Central` and `Peripheral` parameters.

```
estimatedParam(1).CategoryVariableName = 'Age';
estimatedParam(2).CategoryVariableName = 'Age';
estimatedParam(3).CategoryVariableName = 'Sex';
estimatedParam(4).CategoryVariableName = 'Sex';
categoryFit = sbiofit(model, trainData, responseMap, estimatedParam, dose)
```

When fitting by category (or group), `sbiofit` always returns one results object, not one for each category level. This is because both male and female individuals are considered to be part of the same optimization using the same error model and error parameters, similarly for the young and old individuals.

```
categoryFit =
```

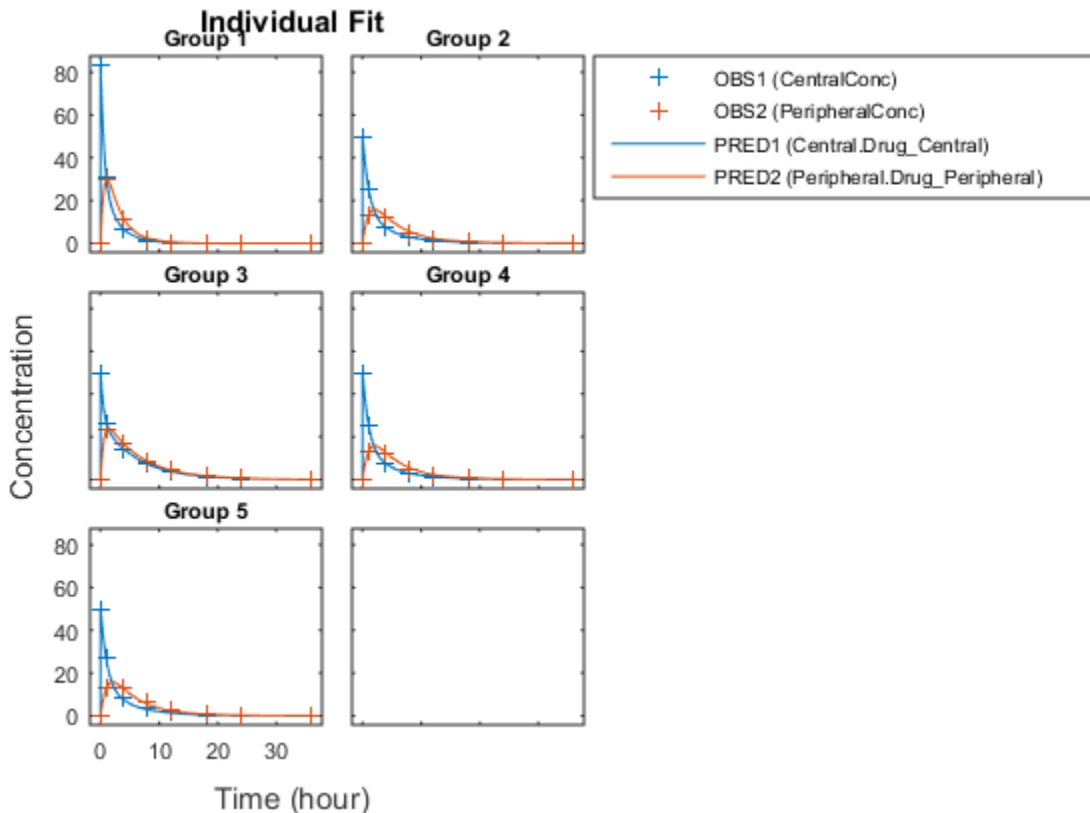
```
    OptimResults with properties:
```

```
        ExitFlag: 3
```

```
        Output: [1x1 struct]
      GroupName: []
        Beta: [8x5 table]
ParameterEstimates: [20x6 table]
          J: [40x8x2 double]
        COVB: [8x8 double]
CovarianceMatrix: [8x8 double]
          R: [40x2 double]
        MSE: 0.1240
        SSE: 8.9269
      Weights: []
EstimatedParameterNames: {'Central' 'Peripheral' 'Q12' 'Cl_Central'}
      ErrorModelInfo: [1x3 table]
EstimationFunction: 'lsqnonlin'
      ErrorModel: 'constant'
      ErrorParameters: [1x1 table]
```

Plot the category-specific estimated results. For the `Cl_Central` and `Q12` parameters, all males had the same estimates, and similarly for the females. For the `Central` and `Peripheral` parameters, all young individuals had the same estimates, and similarly for the old individuals.

```
plot(categoryFit);
```



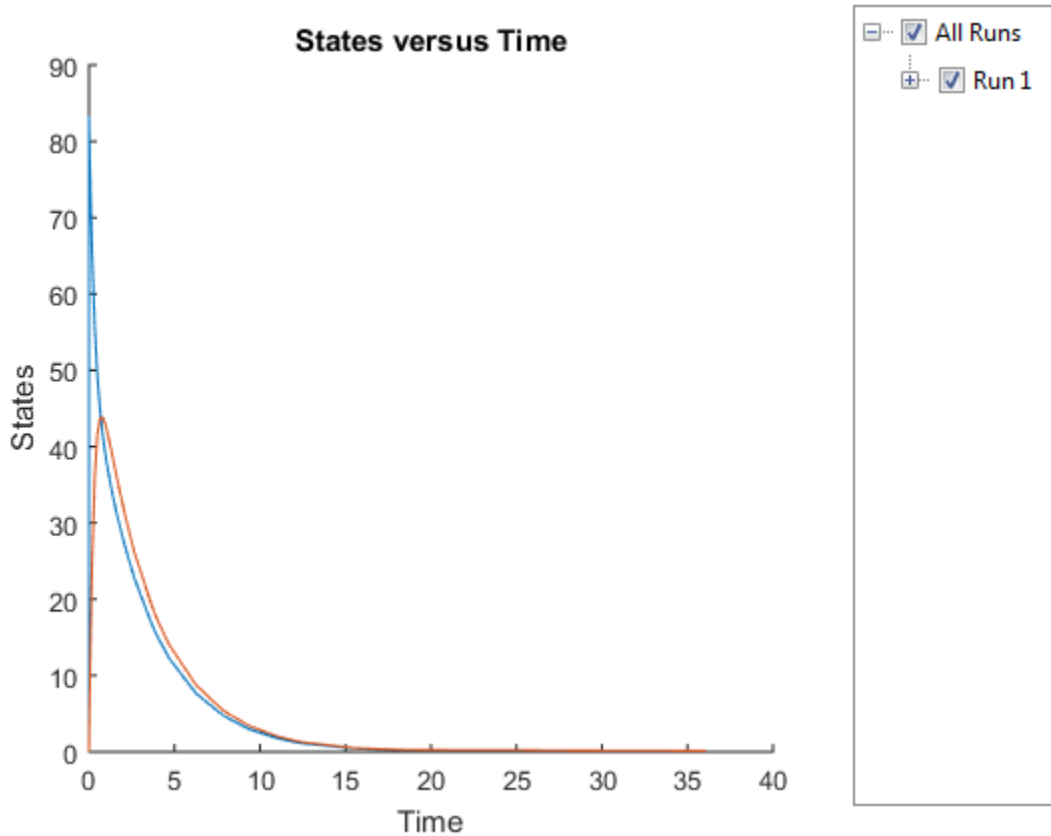
As for testing purposes, simulate the responses of the 6th individual who is an old male. Since you have estimated one set of parameters for the Age category (young versus old), and another set for Sex category (male versus female), you can simulate the responses of an old male even though there is no such individual in the training data.

Use the `predict` method to simulate the responses. `ynew` contains simulation data and `paramestim` contains parameter estimates used during simulation.

```
[ynew,paramestim] = predict(categoryFit,testData,dose);
```

Plot the simulated responses of the old male.

```
sbioplot(ynew);
```



The `paramestim` variable contains the estimated parameters used by the `predict` method. The parameter estimates for corresponding categories were obtained from the `categoryFit.ParameterEstimates` property. Specifically, `Central` and `Peripheral` parameter estimates are obtained from the `Old` group, and `Q12` and `Cl_Central` parameter estimates are obtained from the `Male` group.

`paramestim`

`paramestim =`

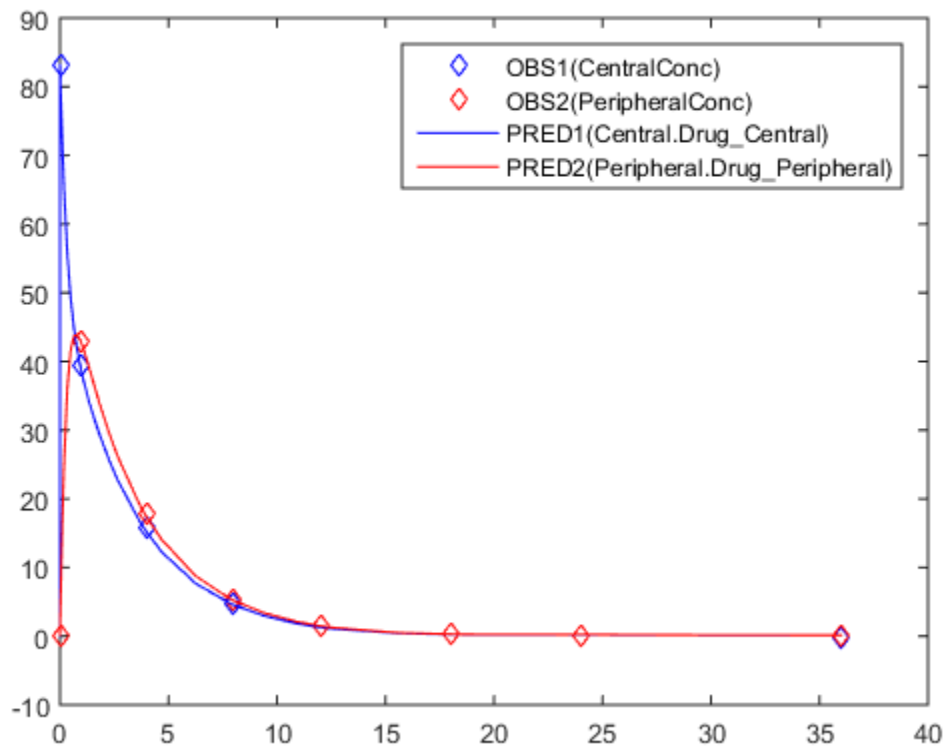
Name	Estimate	StandardError	Group	CategoryVariableName	Category
'Central'	1.1993	0.0050591	6	'Age'	Old



'Peripheral'	0.5627	0.02908	6	'Age'	Old
'Q12'	1.3119	0.062733	6	'Sex'	Male
'Cl_Central'	0.56631	0.0079784	6	'Sex'	Male

Overlay the experimental results on the simulated data.

```
figure;
plot(testData.Time,testData.CentralConc,'LineStyle','none','Marker','d','MarkerEdgeColor','b');
hold on
plot(testData.Time,testData.PeripheralConc,'LineStyle','none','Marker','d','MarkerEdgeColor','r');
plot(ynew.Time,ynew.Data(:,1),'b');
plot(ynew.Time,ynew.Data(:,2),'r');
hold off
legend({'OBS1(CentralConc)','OBS2(PeripheralConc)',...
       'PRED1(Central.Drug\_Central)','PRED2(Peripheral.Drug\_Peripheral)'});
```



### References

- [1] Yi, T-M., Kitano, H., and Simon, M. (2003). A quantitative characterization of the yeast heterotrimeric G protein cycle. PNAS. 100, 10764-10769.

### See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## predict(NLMEResults)

Simulate and evaluate fitted SimBiology model

### Syntax

```
[ynew,parameterEstimates] = predict(resultsObj)
[ynew,parameterEstimates] = predict(resultsObj,data,dosing)
[ynew,parameterEstimates] = predict( ____, 'ParameterType',value)
```

### Description

`[ynew,parameterEstimates] = predict(resultsObj)` returns simulation results `ynew` from evaluating the fitted SimBiology model. The estimated parameter values `parameterEstimates` used to compute `ynew` are from the original fit by `sbiofitmixed`.

`[ynew,parameterEstimates] = predict(resultsObj,data,dosing)` returns simulation results `ynew` from evaluating the fitted SimBiology model by using the specified data and dosing information.

`[ynew,parameterEstimates] = predict( ____, 'ParameterType',value)` returns simulation results `ynew` from evaluating the fitted SimBiology model using either individual or population parameter estimates. The two choices for `value` are 'individual' (default) or 'population'.

---

**Tip** Use this method to get model responses at specific time points or to predict model responses using different covariate data and dosing information.

---

## Examples

### Perform Nonlinear Mixed-Effects Estimation

Estimate nonlinear mixed-effects parameters using clinical pharmacokinetic data collected from 59 infants. Evaluate the fitted model given new data or dosing information.

### Load Data

This example uses data collected on 59 preterm infants given phenobarbital during the first 16 days after birth [1]. `ds` is a table containing the concentration-time profile data and covariate information for each infant (or group).

```
load pheno.mat ds
```

### Convert to groupedData

Convert the data to the `groupedData` format for parameter estimation.

```
data = groupedData(ds);
```

Display the first few rows of `data`.

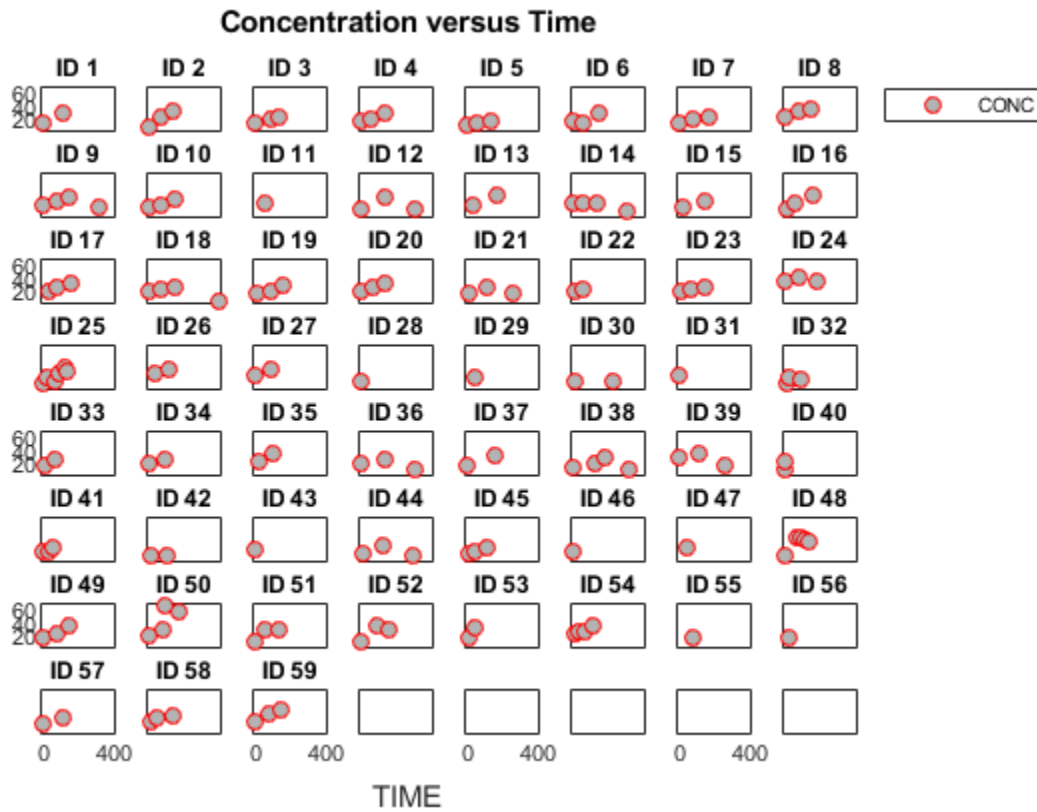
```
data(1:5, :)
```

```
ans=5x6 table
   ID   TIME   DOSE   WEIGHT   APGAR   CONC
   —   ———   ———   ————   ————   ———
   1     0     25     1.4     7     NaN
   1     2     NaN     1.4     7    17.3
   1    12.5    3.5     1.4     7     NaN
   1    24.5    3.5     1.4     7     NaN
   1    37     3.5     1.4     7     NaN
```

### Visualize Data

Display the data in a trellis plot.

```
t = sbiotrellis(data, 'ID', 'TIME', 'CONC', 'marker', 'o', ...
    'markerfacecolor', [.7 .7 .7], 'markeredgecolor', 'r', ...
    'linestyle', 'none');
t.plottitle = 'Concentration versus Time';
```



### Create a One-Compartment PK Model

Create a simple one-compartment PK model, with bolus dose administration and linear clearance elimination, to fit the data.

```
pkmd = PKModelDesign;
addCompartment(pkmd, 'Central', 'DosingType', 'Bolus', ...
               'EliminationType', 'linear-clearance', ...
               'HasResponseVariable', true, 'HasLag', false);
onecomp = pkmd.construct;
```

Map model species to response data.

```
responseMap = 'Drug_Central = CONC';
```

### Define Estimated Parameters

The parameters to estimate in this model are the volume of the central compartment (`Central`) and the clearance rate (`Cl_Central`). `sbiofitmixed` calculates fixed and random effects for each parameter. The underlying algorithm computes normally distributed random effects, which might violate constraints for biological parameters that are always positive, such as volume and clearance. Therefore, specify a transform for the estimated parameters so that the transformed parameters follow a normal distribution. The resulting model is

$$\log(V_i) = \log(\phi_{V,i}) = \theta_V + \eta_{V,i}$$

and

$$\log(Cl_i) = \log(\phi_{Cl,i}) = \theta_{Cl} + \eta_{Cl,i}$$

where  $\theta$ ,  $\eta$ , and  $\phi$  are the fixed effects, random effects, and estimated parameter values respectively, calculated for each infant (group)  $i$ . Some arbitrary initial estimates for  $V$  (volume of central compartment) and  $Cl$  (clearance rate) are used here in the absence of better empirical data.

```
estimatedParams = estimatedInfo({'log(Central)', 'log(Cl_Central)'}, 'InitialValue', [1 1])
```

### Define Dosing

All infants were given the drug, represented by the `Drug_Central` species, where the dosing schedule varies among infants. The amount of drug is listed in the data variable `DOSE`. You can automatically generate dose objects from the data and use them during fitting. In this example, `Drug_Central` is the target species that receives the dose.

```
sampleDose = sbiodose('sample', 'TargetName', 'Drug_Central');  
doses = createDoses(data, 'DOSE', '', sampleDose);
```

### Fit the Model

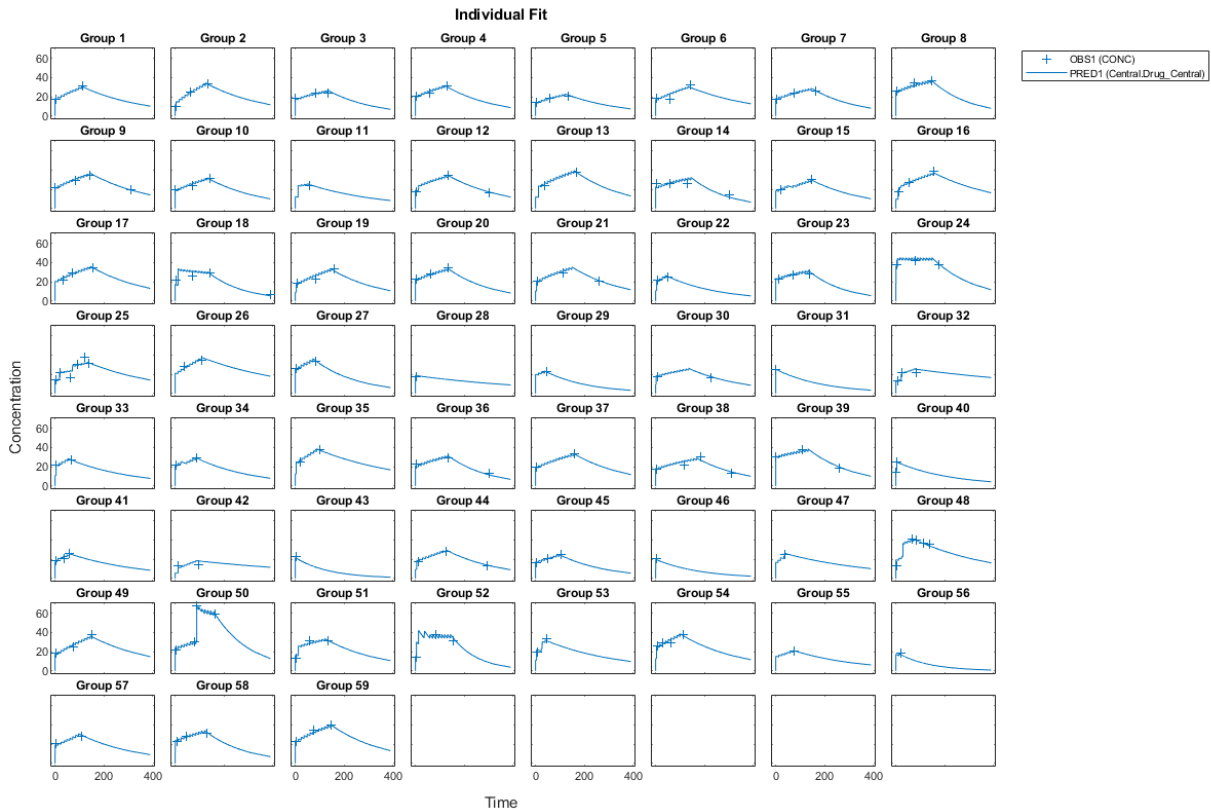
Use `sbiofitmixed` to fit the one-compartment model to the data.

```
nlmeResults = sbiofitmixed(onecomp, data, responseMap, estimatedParams, doses, 'nlmefit');
```

### Visualize Results

Visualize the fitted results using individual-specific parameter estimates.

```
fig1 = plot(nlmeResults, 'ParameterType', 'individual');
% Resize the figure.
fig1.hFig.Position(:) = [100 100 1200 800];
```



### Use New Dosing Data to Simulate the Fitted Model

Suppose you want to predict how infants 1 and 2 would have responded under different dosing amounts. You can predict their responses as follows.

Create new dose objects with new dose amounts.

```
dose1 = doses(1);
dose1.Amount = dose1.Amount*2;
dose2 = doses(2);
dose2.Amount = dose2.Amount*1.5;
```

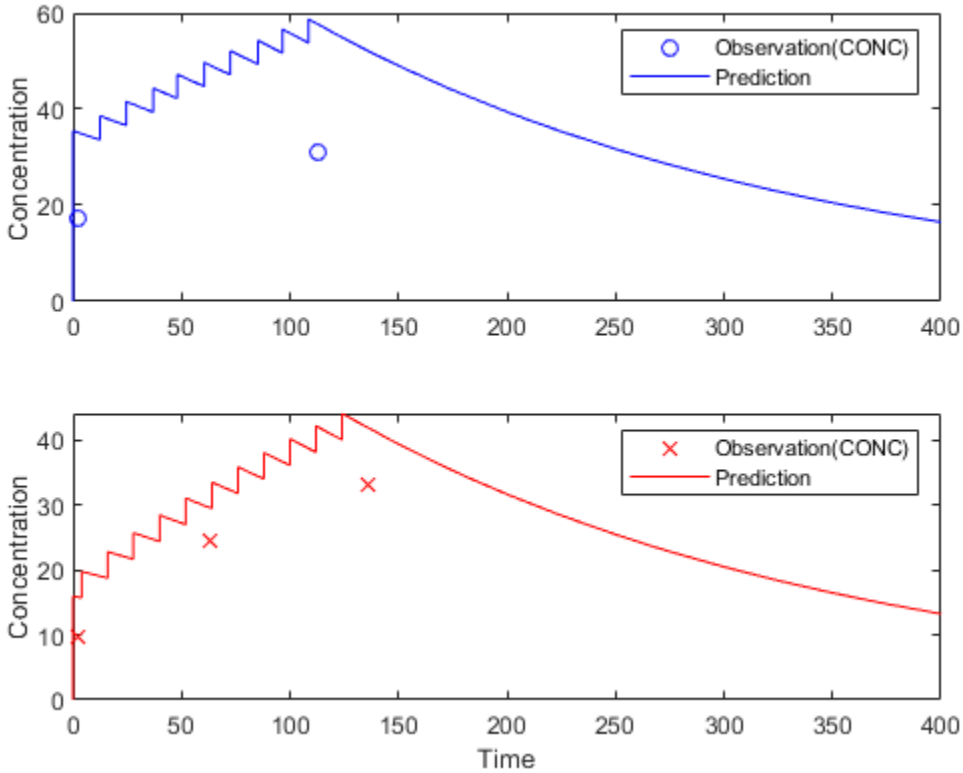
Use the `predict` function to evaluate the fitted model using the new dosing data. If you want response predictions at particular times, provide the new output time vector. Use the 'ParameterType' option to specify individual or population parameters to use. By default, `predict` uses the population parameters when you specify output times.

```
timeVec = [0:25:400];  
newResults = predict(nlmeResults,timeVec,[dose1;dose2],'ParameterType','population');
```

Visualize the predicted responses while overlapping the experimental data for infants 1 and 2.

```
subplot(2,1,1)  
plot(data.TIME(data.ID == 1),data.CONC(data.ID == 1),'bo')  
hold on  
plot(newResults(1).Time,newResults(1).Data,'b')  
hold off  
ylabel('Concentration')  
legend('Observation(CONC)','Prediction')  
subplot(2,1,2)  
plot(data.TIME(data.ID == 2),data.CONC(data.ID == 2),'rx')  
hold on  
plot(newResults(2).Time,newResults(2).Data,'r')  
hold off  
legend('Observation(CONC)','Prediction')  
ylabel('Concentration')  
xlabel('Time')
```





### Create a Covariate Model for the Covariate Dependencies

Suppose there is a correlation between volume and weight, and possibly volume and APGAR score. Consider the effect of weight by modeling two of these covariate dependencies: the volume of central (Central) and the clearance rate (Cl\_Central) vary with weight. The model becomes

$$\log(V_i) = \log(\phi_{V,i}) = \theta_V + \theta_{V/weight} * weight_i + \eta_{V,i}$$

and

$$\log(Cl_i) = \log(\phi_{Cl,i}) = \theta_{Cl} + \theta_{Cl/weight} * weight_i + \eta_{Cl,i}$$

Use the `CovariateModel` object to define the covariate dependencies. For details, see “Specify a Covariate Model”.

```
covModel = CovariateModel;
covModel.Expression = ({'Central = exp(theta1 + theta2*WEIGHT + eta1)',...
                       'Cl_Central = exp(theta3 + theta4*WEIGHT + eta2)'});
% Use |constructDefaultInitialEstimate| to create an |initialEstimates|
% struct.
initialEstimates = covModel.constructDefaultFixedEffectValues;
```

Use the `FixedEffectNames` property to display the thetas (fixed effects) defined in the model.

```
covModel.FixedEffectNames

ans = 4x1 cell array
    {'theta1'}
    {'theta3'}
    {'theta2'}
    {'theta4'}
```

Use the `FixedEffectDescription` property to show the descriptions of corresponding fixed effects (thetas) used in the covariate expression. For example, `theta2` is the fixed effect for the weight covariate that correlates with the volume (`Central`), denoted as `'Central/WEIGHT'`.

```
disp('Fixed Effects Description:');

Fixed Effects Description:

disp(covModel.FixedEffectDescription);

    'Central'
    'Cl_Central'
    'Central/WEIGHT'
    'Cl_Central/WEIGHT'
```

Set the initial guesses for the fixed-effect parameter values for `Central` and `Cl_Central` using the values estimated from fitting the base model.

```
initialEstimates.theta1 = nlmeResults.FixedEffects.Estimate(1);
initialEstimates.theta3 = nlmeResults.FixedEffects.Estimate(2);
covModel.FixedEffectValues = initialEstimates;
```

**Fit the Model**

```
nlmeResults_cov = sbiofitmixed(onecomp,data,responseMap,covModel,doses,'nlmefit');
```

**Display Fitted Parameters and Covariances**

```
disp('Estimated Fixed Effects:');
```

```
Estimated Fixed Effects:
```

```
disp(nlmeResults_cov.FixedEffects);
```

Name	Description	Estimate	StandardError
'theta1'	'Central'	-0.45664	0.078933
'theta3'	'CL_Central'	-5.9519	0.1177
'theta2'	'Central/WEIGHT'	0.52948	0.047342
'theta4'	'CL_Central/WEIGHT'	0.61954	0.071386

```
disp('Estimated Covariance Matrix:');
```

```
Estimated Covariance Matrix:
```

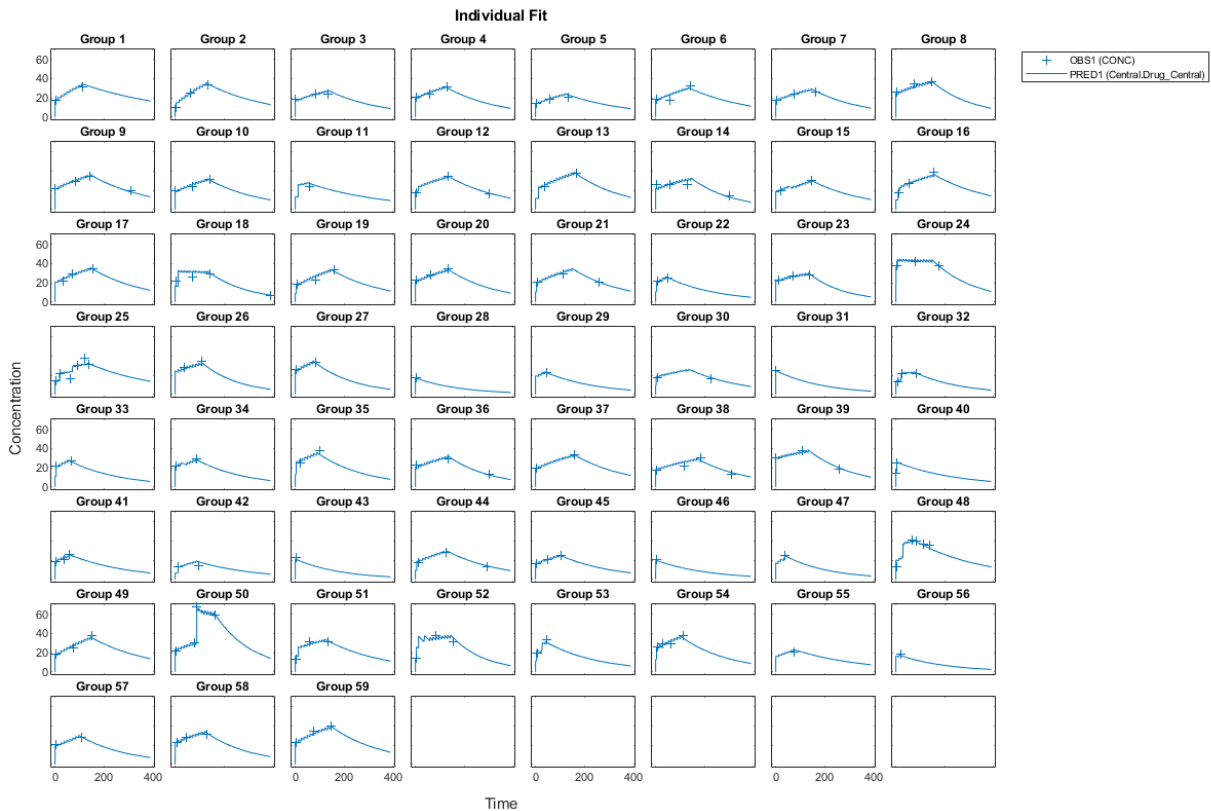
```
disp(nlmeResults_cov.RandomEffectCovarianceMatrix);
```

	eta1	eta2
eta1	0.046503	0
eta2	0	0.041609

**Visualize Results**

Visualize the fitted results using individual-specific parameter estimates.

```
fig2 = plot(nlmeResults_cov,'ParameterType','individual');
% Resize the figure.
fig2.hFig.Position(:) = [100 100 1200 800];
```



### Use New Covariate Data to Evaluate the Fitted Model

Suppose you want to explore the responses of infants 1 and 2 using different covariate data, namely `WEIGHT`. You can do this by specifying the new `WEIGHT` data. The `ID` variable of the data corresponds to individual infants.

```
newData = data(data.ID == 1 | data.ID == 2, :);
newData.WEIGHT(newData.ID == 1) = 1.3;
newData.WEIGHT(newData.ID == 2) = 1.4;
```

Simulate the responses of infants 1 and 2 using the new covariate data.

```
[newResults_cov, newEstimates] = predict(nlmeResults_cov, newData, [dose1; dose2]);
```

`newEstimates` contains the updated parameter estimates for each individual (infants 1 and 2) after the model is reevaluated using the new covariate data.

`newEstimates`

```
newEstimates=4x3 table
  Group      Name      Estimate
  -----
  1      'Central'      2.5596
  1      'Cl_Central'    0.0065965
  2      'Central'      1.7123
  2      'Cl_Central'    0.0064806
```

Compare to the estimated values from the original fit using the old covariate data.

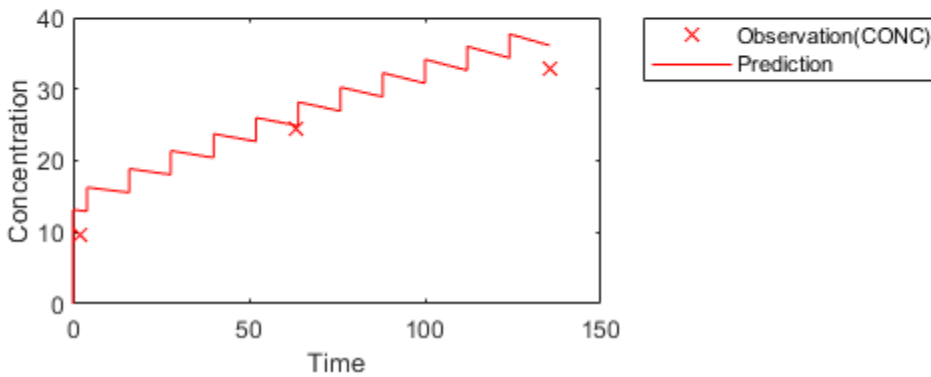
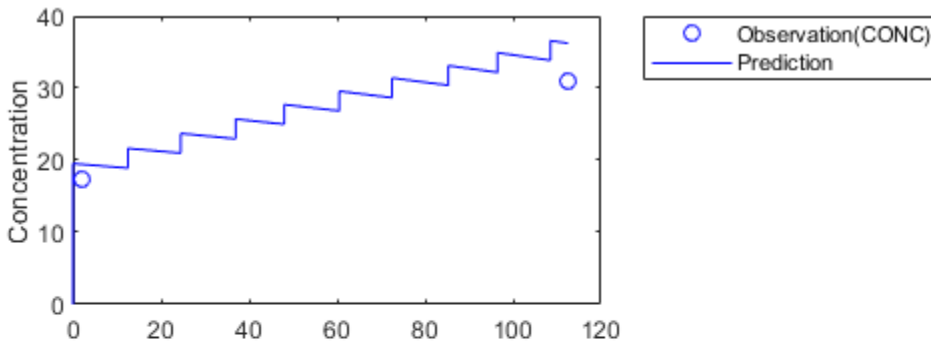
```
nlmeResults_cov.IndividualParameterEstimates( ...
  nlmeResults_cov.IndividualParameterEstimates.Group == '1' | ...
  nlmeResults_cov.IndividualParameterEstimates.Group == '2',:)
```

```
ans=4x3 table
  Group      Name      Estimate
  -----
  1      'Central'      2.6988
  1      'Cl_Central'    0.0070181
  2      'Central'      1.8054
  2      'Cl_Central'    0.0068948
```

Visualize the new simulation results together with the experimental data for infant 1 and 2.

```
subplot(2,1,1)
plot(data.TIME(data.ID == 1),data.CONC(data.ID == 1),'bo')
hold on
plot(newResults_cov(1).Time,newResults_cov(1).Data,'b')
hold off
ylabel('Concentration')
legend('Observation(CONC)', 'Prediction', 'Location', 'NorthEastOutside')
subplot(2,1,2)
plot(data.TIME(data.ID == 2),data.CONC(data.ID == 2),'rx')
hold on
plot(newResults_cov(2).Time,newResults_cov(2).Data,'r')
```

```
hold off
legend('Observation(CONC)', 'Prediction', 'Location', 'NorthEastOutside')
ylabel('Concentration')
xlabel('Time')
```



## References

[1] Grasela, T. H. Jr., and S. M. Donn. "Neonatal population pharmacokinetics of phenobarbital derived from routine clinical data." *Dev Pharmacol Ther* 1985;8(6). 374-83. PubMed Abstract

## Input Arguments

### **resultsObj** — Estimation results

scalar `NLMEResults` object

Estimation results, specified as a scalar `NLMEResults` object, which contains nonlinear mixed-effects estimation results returned by `sbiofitmixed`.

### **data** — Grouped data or output times

`groupedData` object | vector | cell array of vectors

Grouped data or output times, specified as a `groupedData` object, vector, or cell array of vectors of output times.

If `data` is a `groupedData` object, it must have both group labels and output times specified. The group labels can refer to new groups or existing groups from the original fit. If the mixed-effects model from the original fit (returned by `sbiofitmixed`) uses covariates, the `groupedData` object must also contain the covariate data with the same labels for the covariates (`CovariateLabels` property) specified in the original covariate model on page 2-98.

By default, individual parameter estimates are used for simulating groups from the original fit, while population parameters are used for new groups, if any. See the `value` argument description for details.

The total number of simulation results in the output `ynew` depends on the number of groups or output time vectors in `data` and the number of rows in the `dosing` matrix. For details, see the table in the `ynew` argument description.

### **dosing** — Dosing information

[ ] | 2-D matrix of `SimBiology` dose objects

Dosing information, specified as an empty array [ ] or 2-D matrix of `SimBiology` dose objects (`ScheduleDose` object or `RepeatDose` object). If `dosing` is a matrix of dose

objects, the matrix must contain default doses or be consistent with the original dosing data used with `sbiofitmixed`. That is, dose objects in `dosing` must have the same values for dose properties (such as `TargetName`) or must be parameterized in the same way as the original dosing data. For instance, suppose that the original dosing matrix has two columns of doses, where the first column targets species  $x$  and the second column targets species  $y$ . Then `dosing` must have doses in the first column targeting species  $x$  and doses in the second column targeting species  $y$ .

- If empty, no doses are applied during simulation, even if the model has active doses.
- If not empty, the matrix must have a single row or one row per group in the input data. If it has a single row, the same doses are applied to all groups during simulation. If it has multiple rows, each row is applied to a separate group, in the same order as the groups appear in the input data. If some groups (or time vectors) require more doses than others, then fill in the matrix with default (dummy) doses.
- Multiple columns are allowed so that you can apply multiple dose objects to each group or time vector. All doses in a column must be default doses or must reference the same components in the model (for instance, the doses must have the same `TargetName`) and must have consistent parameterized properties as in the original dosing data used with `sbiofitmixed`. For example, if the `Amount` property of a dose used in the original `sbiofitmixed` call is parameterized to a model-scoped parameter 'A', all doses for the corresponding group (column) in `dosing` must have the `Amount` property parameterized to 'A'.
- A default dose has default values for all properties, except for the `Name` property. Create a default dose as follows.

```
d1 = sbiodose('d1');
```
- In addition to manually constructing dose objects using `sbiodose`, if the input data is a `groupedData` object and has dosing information, you can use the `createDoses` method to construct doses from it.

The number of rows in the `dosing` matrix and the number of groups or output time vectors in `data` determine the total number of simulation results in the output `ynew`. For details, see the table in the `ynew` argument description.

---

**Note** If `UnitConversion` is turned on for the underlying SimBiology model that was used for fitting, `dosing` must specify valid amount and time units.

---



**value — Parameter type**`'individual' (default) | 'population'`

Parameter type, specified as `'individual'` (default) or `'population'`. If `value` is `'population'`, the `predict` method returns the simulation results using the population parameter estimates, that is, parameter values that are estimated using fixed effects ( $\theta$ s) only. The estimated parameter values used in simulation are identical to those in the `resultsObj.PopulationParameterEstimates` property, unless you specify a new `groupedData` object `data` with new covariate data. In this case, the method reevaluates the covariate model and the parameter estimates based on the new `groupedData` and covariate data.

If `value` is `'individual'`, the method returns the simulation results using the corresponding parameter values of the group in the `resultsObj.IndividualParameterEstimates` property. These values include both fixed- and random-effects estimates, that is, parameter values estimated using both fixed effects ( $\theta$ s) and random effects ( $\eta$ s). If `data` contains new groups, only fixed effects (population parameter estimates of the results object) are used for these groups.

By default, `predict` uses the individual parameter estimates of the results object when `data` is a `groupedData` object. If `data` is a vector of output times or cell array of vectors, `predict` uses the population parameter estimates of the results object instead.

## Output Arguments

**ynew — Simulation results**vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects. The states reported in `ynew` are the states included in the `responseMap` input argument of `sbiofitmixed` and any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the `SimBiology` model.

The total number of simulation results in `ynew` depends on the number of groups or output time vectors in `data` and the number of rows in the `dosing` matrix.

Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
1	0, that is, dosing is empty [ ]	The total number of SimData objects in ynew is 1.  No doses are applied during simulation.
1	1	The total number of SimData objects in ynew is 1.  The given row of doses is applied during the simulation.
1	<i>N</i>	The total number of SimData objects in ynew is <i>N</i> .  Each row of dosing is applied to each simulation.
<i>N</i>	0, that is, dosing is empty [ ]	The total number of SimData objects in ynew is <i>N</i> .  No doses are applied during simulation.
<i>N</i>	1	The total number of SimData objects in ynew is <i>N</i> .  The same row of doses is applied to each simulation.

Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
$N$	$N$	The total number of SimData objects in ynew is $N$ .  Each row of dosing is applied to a separate group, in the same order that groups appear in data.
$M$	$N$	The function throws an error when $M \neq N$ .

### parameterEstimates – Estimated parameter values

table

Estimated parameter values used for the predicted simulation results, returned as a table.

If 'ParameterType' is 'individual', the reported parameter values are identical to the values in the resultsObj.IndividualParameterEstimates property. However, if data contains new groups, then only population parameter estimates (fixed effects) are used for these groups. The corresponding reported values in parameterEstimates for these groups are identical to the values in resultsObj.PopulationParameterEstimates.

If 'ParameterType' is 'population', the reported parameter values are identical to the values in the resultsObj.PopulationParameterEstimates property unless you specify new covariate information in data. See the value argument description for details.

If data is a vector or a cell array of vectors of output times, the reported parameter values are identical to the values in resultsObj.PopulationParameterEstimates. Also, the groups reported represent the enumeration of simulations performed and are unrelated to group names in the original fit.

## See Also

CovariateModel object | NLMEResults object | sbiofitmixed | sbiosampleerror | sbiosampleparameters

## **Topics**

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”

“Nonlinear Mixed-Effects Modeling”

**Introduced in R2014a**

# random(LeastSquaresResults, OptimResults, NLINResults)

Simulate SimBiology model, adding variations by sampling error model

## Syntax

```
[ynew, parameterEstimates] = random(resultsObj)  
[ynew, parameterEstimates] = random(resultsObj, data, dosing)
```

## Description

[ynew, parameterEstimates] = random(resultsObj) returns simulation results ynew with added noise using the error model information specified by the resultsObj.ErrorModelInfo property and estimated parameter values parameterEstimates.

[ynew, parameterEstimates] = random(resultsObj, data, dosing) uses the specified data and dosing information.

---

**Note** The noise is only added to states that are responses which are the states included in the responseMap input argument when you called sbiofit. If there is a separate error model for each response, the noise is added to each response separately using the corresponding error model.

---

## Input Arguments

### resultsObj — Estimation results

OptimResults object | NLINResults object

Estimation results, specified as an OptimResults object or NLINResults object, which contains estimation results returned by sbiofit. It must be a scalar object.

### **data — Grouped data or output times**

`groupedData` object | vector | cell array of vectors

Grouped data or output times, specified as a `groupedData` object, vector, or cell array of vectors of output times.

If it is a vector of time points, `random` simulates the model with new time points using the parameter estimates from the results object `resultsObj`.

If it is a cell array of vectors of time points, `random` simulates the model  $n$  times using the output times from each time vector, where  $n$  is the length of `data`.

If it is a `groupedData` object, it must have an independent variable such as `Time`. It must also have a group variable if the training data used for fitting has such variable. You can use a `groupedData` object to query different combinations of categories if the `resultsObj` contains parameter estimates for each category. `random` simulates the model for each group with the specified categories. For instance, suppose you have a set of parameter estimates for sex category (males versus females), and age category (young versus old) in your training data. You can use `random` to simulate the responses of an old male (or any other combination) although such patient may not exist in the training data.

If the `resultsObj` is from estimating category-specific parameters, `data` must be a `groupedData` object.

---

**Note** If `UnitConversion` is turned on for the underlying SimBiology model that was used for fitting and `data` is a `groupedData` object, `data` must specify valid variable units via `data.Properties.VariableUnits` property. If it is a numeric vector or cell array of vectors of time points, `random` uses the model's `TimeUnits`.

---

### **dosing — Dosing information**

`[]` | 2-D matrix of SimBiology dose objects

Dosing information, specified as an empty array `[]` or 2-D matrix of SimBiology dose objects (`ScheduleDose` object or `RepeatDose` object). If `dosing` is a matrix of dose objects, the matrix must contain default doses or be consistent with the original dosing data used with `sbiofit`. That is, dose objects in `dosing` must have the same values for dose properties (such as `TargetName`) or must be parameterized in the same way as the original dosing data. For instance, suppose that the original dosing matrix has two columns of doses, where the doses in the first column target species  $x$  and those in the

second column target species  $y$ . Then `dosing` must have doses in the first column targeting species  $x$  and those in the second column targeting species  $y$ .

- If empty, no doses are applied during simulation, even if the model has active doses.
- If not empty, the matrix must have a single row or one row per group in the input data. If it has a single row, the same doses are applied to all groups during simulation. If it has multiple rows, each row is applied to a separate group, in the same order as the groups appear in the input data. If some groups (or time vectors) require more doses than others, then fill in the matrix with default (dummy) doses.
- Multiple columns are allowed so that you can apply multiple dose objects to each group or time vector. All doses in a column must be default doses or must reference the same components in the model (for instance, the doses must have the same dose target `TargetName`) and must have consistent parameterized properties as in the original dosing data used with `sbiofit`. For example, if the `Amount` property of a dose used in the original `sbiofit` call is parameterized to a model-scoped parameter 'A', all doses for the corresponding group (column) in `dosing` must have the `Amount` property parameterized to 'A'.
- A default dose has default values for all properties, except for the `Name` property. Create a default dose as follows.

```
d1 = sbiodose('d1');
```

- In addition to manually constructing dose objects using `sbiodose`, if the input data is a `groupedData` object and has dosing information, you can use the `createDoses` method to construct doses from it.

The number of rows in the `dosing` matrix and the number of groups or output time vectors in `data` determine the total number of simulation results in the output `ynew`. For details, see the table in the `ynew` argument description.

---

**Note** If `UnitConversion` is turned on for the underlying SimBiology model that was used for fitting, `dosing` must specify valid amount and time units.

---

## Output Arguments

### **ynew** — Simulation results with noise

vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects. The states reported in `ynew` are the states that were included in the `responseMap` input argument of `sbiofit` as

well as any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the `SimBiology` model.

The total number of simulation results in `ynew` depends on the number of groups or output time vectors in `data` and the number of rows in the `dosing` matrix.

Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
1	0, that is, <code>dosing</code> is empty []	The total number of <code>SimData</code> objects in <code>ynew</code> is 1.  No doses are applied during simulation.
1	1	The total number of <code>SimData</code> objects in <code>ynew</code> is 1.  The given row of doses is applied during the simulation.
1	$N$	The total number of <code>SimData</code> objects in <code>ynew</code> is $N$ .  Each row of <code>dosing</code> is applied to each simulation.
$N$	0, that is, <code>dosing</code> is empty []	The total number of <code>SimData</code> objects in <code>ynew</code> is $N$ .  No doses are applied during simulation.



Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
$N$	1	The total number of SimData objects in ynew is $N$ .  The same row of doses is applied to each simulation.
$N$	$N$	The total number of SimData objects in ynew is $N$ .  Each row of dosing is applied to a separate group, in the same order that groups appear in data.
$M$	$N$	The function throws an error when $M \neq N$ .

### parameterEstimates — Estimated parameter values

table

Estimated parameter values, returned as a table. This is identical to resultsObj.ParameterEstimates property.

## Examples

### Add Noise to Simulation Results of a Fitted SimBiology Model

This example uses the yeast heterotrimeric G protein model and experimental data reported by [1]. For more details about the model, see the **Background** section in “Parameter Scanning, Parameter Estimation, and Sensitivity Analysis in the Yeast Heterotrimeric G Protein Cycle”.

Load the G protein model.

```
sbioloadproject gprotein
```

Enter the experimental data containing the time course for the fraction of active G protein, as reported in the reference paper [1].

```
time = [0 10 30 60 110 210 300 450 600]';  
GaFracExpt = [0 0.35 0.4 0.36 0.39 0.33 0.24 0.17 0.2]';
```

Create a `groupedData` object based on the experimental data.

```
tbl = table(time,GaFracExpt);  
grpData = groupedData(tbl);
```

Map the appropriate model component to the experimental data. In other words, indicate which species in the model corresponds to which response variable in the data. In this example, map the model parameter `GaFrac` to the experimental data variable `GaFracExpt` from `grpData`.

```
responseMap = 'GaFrac = GaFracExpt';
```

Use an `estimatedInfo` object to define the model parameter `kGd` as a parameter to be estimated.

```
estimatedParam = estimatedInfo('kGd');
```

Perform the parameter estimation. Use the name-value pair argument `'ErrorModel'` to specify the error model that adds error to simulation data.

```
fitResult = sbiofit(m1,grpData,responseMap,estimatedParam,'ErrorModel','proportional');
```

View the estimated parameter value of `kGd`.

```
fitResult.ParameterEstimates
```

```
ans =
```

Name	Estimate	StandardError
'kGd'	0.11	0.00064116

Use the `random` method to retrieve the simulation data with added noise using the proportional error model which was specified by `sbiofit`. Note that the noise is added only to the response state, that is the `GaFrac` parameter.

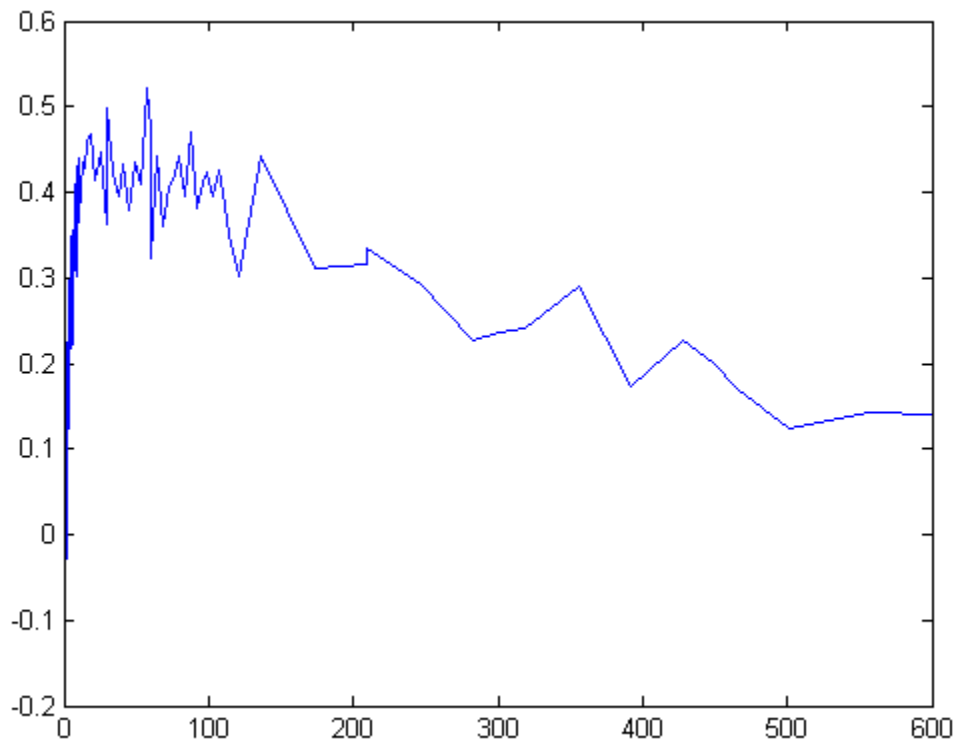
```
[ynew,paramEstim] = random(fitResult);
```

Select the simulation data for the GaFrac parameter.

```
GaFracNew = select(ynew, {'Name', 'GaFrac'});
```

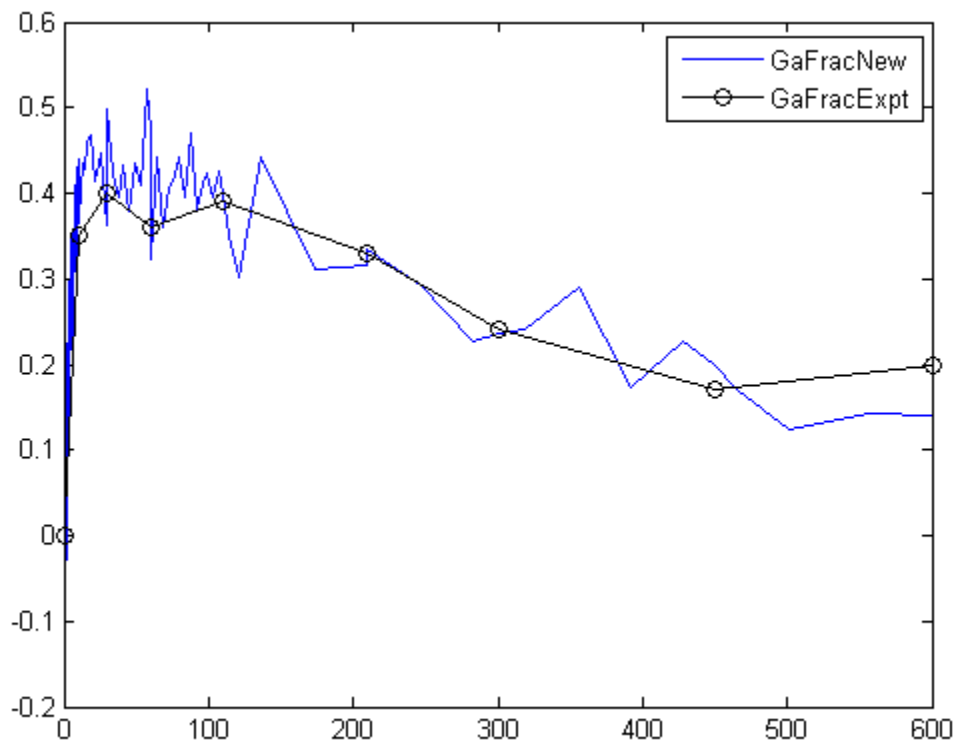
Plot the simulation results.

```
plot(GaFracNew.Time, GaFracNew.Data)  
hold on
```



Plot the experimental data to compare it with the simulated data.

```
plot(time, GaFracExpt, 'Color', 'k', 'Marker', 'o')  
legend('GaFracNew', 'GaFracExpt')
```



### References

- [1] Yi, T-M., Kitano, H., and Simon, M. (2003). A quantitative characterization of the yeast heterotrimeric G protein cycle. PNAS. 100, 10764-10769.

### See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## random(NLMEResults)

Simulate a SimBiology model, adding variations by sampling the error model

### Syntax

```
[ynew,parameterEstimates,randomEffects]= random(resultsObj)
[ynew,parameterEstimates]= random(resultsObj,data,dosing)
[ynew,parameterEstimates,randomEffects]=
random(_, 'ParameterType',value)
```

### Description

[ynew,parameterEstimates,randomEffects]= random(resultsObj) returns simulation results ynew with added noise using the error model information specified by the resultsObj.ErrorModelInfo property and estimated parameter values parameterEstimates which are returned by sbiofitmixed.

[ynew,parameterEstimates]= random(resultsObj,data,dosing) uses the specified data and dosing information.

[ynew,parameterEstimates,randomEffects]= random(\_, 'ParameterType',value) adds noise to simulation results that are simulated using either individual or population parameter estimates. The two choices for value are 'population' or 'individual' (default).

---

**Note** The noise is only added to states that are responses which are the states included in the responseMap input argument when you called sbiofitmixed.

---

### Input Arguments

**resultsObj** — Estimation results

NLMEResults object

Estimation results, specified as an `NLMEResults` object, which contains estimation results returned by `sbiofitmixed`. It must be a scalar object.

### **data** — Grouped data or output times

`groupedData` object | vector | cell array of vectors

Grouped data or output times, specified as a `groupedData` object, vector, or cell array of vectors of output times.

If it is a vector of time points, `random` simulates the model with new time points.

If it is a cell array of vectors of time points, `random` simulates the model  $n$  times using the output times from each time vector, where  $n$  is the length of `data`.

For both cases, new parameter values are calculated using `sbiosampleparameters` with the covariate model returned by `resultsObj.covariateModel`, the fixed effect estimates (`resultsObj.FixedEffects`), and random effect covariance matrix (`resultsObj.RandomEffectCovarianceMatrix`).

If the mixed-effects model from the original fit (using `sbiofitmixed`) uses a covariate model with covariates, the `data` must be a `groupedData` object containing covariate data with the same labels for the covariates (`CovariateLabels` property) specified in the original covariate model.

### **dosing** — Dosing information

`[]` | 2-D matrix of SimBiology dose objects

Dosing information, specified as an empty array `[]` or 2-D matrix of SimBiology dose objects (`ScheduleDose` object or `RepeatDose` object). If `dosing` is a matrix of dose objects, the matrix must contain default doses or be consistent with the original dosing data used with `sbiofitmixed`. That is, dose objects in `dosing` must have the same values for dose properties (such as `TargetName`) or must be parameterized in the same way as the original dosing data. For instance, suppose that the original dosing matrix has two columns of doses, where the first column targets species  $x$  and the second column targets species  $y$ . Then `dosing` must have doses in the first column targeting species  $x$  and doses in the second column targeting species  $y$ .

- If empty, no doses are applied during simulation, even if the model has active doses.
- If not empty, the matrix must have a single row or one row per group in the input data. If it has a single row, the same doses are applied to all groups during simulation. If it has multiple rows, each row is applied to a separate group, in the same order as the

groups appear in the input data. If some groups (or time vectors) require more doses than others, then fill in the matrix with default (dummy) doses.

- Multiple columns are allowed so that you can apply multiple dose objects to each group or time vector. All doses in a column must be default doses or must reference the same components in the model (for instance, the doses must have the same `TargetName`) and must have consistent parameterized properties as in the original dosing data used with `sbiofitmixed`. For example, if the `Amount` property of a dose used in the original `sbiofitmixed` call is parameterized to a model-scoped parameter 'A', all doses for the corresponding group (column) in `dosing` must have the `Amount` property parameterized to 'A'.
- A default dose has default values for all properties, except for the `Name` property. Create a default dose as follows.

```
d1 = sbiodose('d1');
```

- In addition to manually constructing dose objects using `sbiodose`, if the input data is a `groupedData` object and has dosing information, you can use the `createDoses` method to construct doses from it.

The number of rows in the `dosing` matrix and the number of groups or output time vectors in `data` determine the total number of simulation results in the output `ynew`. For details, see the table in the `ynew` argument description.

---

**Note** If `UnitConversion` is turned on for the underlying SimBiology model that was used for fitting, `dosing` must specify valid amount and time units.

---

### value — Parameter type

'individual' (default) | 'population'

Parameter type, specified as 'population' or 'individual' (default). If `value` is 'population', the method returns the simulation results with noise using the population parameter estimates. The estimated parameter values used in simulation are identical to `resultsObj.PopulationParameterEstimates` property unless you specify a new `groupedData` object `data` with new covariate data. In this case, the method will reevaluate the covariate model and this could change the parameter estimates.

If `value` is 'individual', estimated parameter values and random effect values are resampled by calling `sbiosampleparameters` with the covariate model (specified by the `data` argument or returned by the `covariateModel` method of `resultsObj`), the fixed

effect estimates (`resultsObj.FixedEffects`), and random effect covariance matrix (`resultsObj.RandomEffectCovarianceMatrix`). Parameter estimates and random effects are resampled for all groups.

## Output Arguments

### **ynew** — Simulation results with noise

vector of `SimData` objects

Simulation results, returned as a vector of `SimData` objects. The states reported in `ynew` are the states included in the `responseMap` input argument of `sbiofitmixed` and any other states listed in the `StatesToLog` property of the runtime options (`RuntimeOptions`) of the `SimBiology` model.

The total number of simulation results in `ynew` depends on the number of groups or output time vectors in `data` and the number of rows in the `dosing` matrix.

Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
1	0, that is, dosing is empty []	The total number of <code>SimData</code> objects in <code>ynew</code> is 1.  No doses are applied during simulation.
1	1	The total number of <code>SimData</code> objects in <code>ynew</code> is 1.  The given row of doses is applied during the simulation.



Number of groups or output time vectors in data	Number of rows in the dosing matrix	Simulation results
1	$N$	The total number of SimData objects in ynew is $N$ .  Each row of dosing is applied to each simulation.
$N$	0, that is, dosing is empty []	The total number of SimData objects in ynew is $N$ .  No doses are applied during simulation.
$N$	1	The total number of SimData objects in ynew is $N$ .  The same row of doses is applied to each simulation.
$N$	$N$	The total number of SimData objects in ynew is $N$ .  Each row of dosing is applied to a separate group, in the same order that groups appear in data.
$M$	$N$	The function throws an error when $M \neq N$ .

**parameterEstimates** – Estimated parameter values

table

Estimated parameter values, returned as a table.

If you specify the `value` argument as `'individual'`, these estimated values will differ from those values from the original fit since parameter values are recalculated using `sbiosampleparameters`.

If `'ParameterType'` is `'population'`, the estimated parameter values are identical to `resultsObj.PopulationParameterEstimates` property unless you specify a new `groupedData` object data with new covariate data.

### **randomEffects — Random effect values**

table

Random effect values, specified as a table.

## **See Also**

NLMEResults object | sbiofitmixed | sbiosampleerror | sbiosampleparameters

**Introduced in R2014a**

# ParameterConfidenceInterval

Object containing confidence interval results for estimated parameters

## Description

The `ParameterConfidenceInterval` object contains confidence interval results for the estimated parameters computed using `sbioparameterci`.

## Creation

## Syntax

```
ci = sbioparameterci(fitResults)
ci = sbioparameterci(fitResults,Name,Value)
```

## Description

`ci = sbioparameterci(fitResults)` computes 95% confidence intervals for the estimated parameters from `fitResults`, a `NLINResults` object or `OptimResults` object returned by `sbiofit`. `ci` is a `ParameterConfidenceInterval` object that contains the computed confidence intervals.

`ci = sbioparameterci(fitResults,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### **fitResults** — Parameter estimation results from `sbiofit`

`NLINResults` object | `OptimResults` object | vector

Parameter estimation results from `sbiofit`, specified as an `NLINResults` object, `OptimResults` object, or a vector of objects for unpooled fits that were returned from the same `sbiofit` call.

### **Alpha — Confidence level**

0.05 (default) | positive scalar

Confidence level,  $(1 - \text{Alpha}) * 100\%$ , specified as the comma-separated pair consisting of 'Alpha' and a positive scalar between 0 and 1. The default value is 0.05, meaning a 95% confidence interval is computed.

Example: 'Alpha',0.01

### **Type — Confidence interval type**

'gaussian' (default) | 'profileLikelihood' | 'bootstrap'

Confidence interval type, specified as the comma-separated pair consisting of 'Type' and a character vector. The valid choices are:

- 'gaussian' - Use the Gaussian approximation on page 1-209 of the distribution of parameter estimates.
- 'profileLikelihood' - Compute the profile likelihood intervals. This type is not supported for parameter estimates from hierarchical models, that is, estimated results from fitting different categories on page 1-77 (such as Age or Sex). In other words, if you set the `CategoryVariableName` property of the `EstimatedInfo` object in your original fit, then the fit results are hierarchical, therefore, you cannot compute the `profileLikelihood` confidence intervals on the results. For details, see “Profile Likelihood Confidence Interval Calculation” on page 1-210.
- 'bootstrap' - Compute confidence intervals using the bootstrap method on page 1-212.

Example: 'Type','bootstrap'

### **Tolerance — Tolerance for profile likelihood and bootstrap confidence interval computations**

1e-5 (default) | positive scalar

Tolerance for the profile likelihood and bootstrap confidence interval computations, specified as the comma-separated pair consisting of 'Tolerance' and a positive scalar.

The profile likelihood method uses this value as a termination tolerance. For details, see “Profile Likelihood Confidence Interval Calculation” on page 1-210.

The bootstrap method uses this value to determine whether a confidence interval is constrained by bounds specified in the original fit. For details, see “Bootstrap Confidence Interval Calculation” on page 1-212.

Example: 'Tolerance', 1e-6

### **MaxStepSize** — Maximum step size used for computing profile likelihood curves

0.1 (default) | positive scalar | [] | cell array

Maximum step size used for computing profile likelihood curves, specified as the comma-separated pair consisting of 'MaxStepSize' and a positive scalar, [], or cell array. By default, this argument is set to 0.1. If you set it to [], then the maximum step size is set to 10% of the width of the Gaussian approximation of the confidence interval, if it exists. You can specify a maximum step size (or []) for each estimated parameter using a cell array.

Example: 'MaxStepSize', 0.5

### **NumSamples** — Number of samples for bootstrapping

1000 (default) | positive integer

Number of samples for bootstrapping, specified as the comma-separated pair consisting of 'NumSamples' and a positive integer. This number defines the number of fits that are performed during the confidence interval computation to generate bootstrap samples. The smaller the number is, the faster the computation of the confidence intervals becomes, at the cost of decreased accuracy.

Example: 'NumSamples', 500

### **Display** — Level of display returned to the command line

'off' (default) | 'none' | 'final' | 'iter'

Level of display returned to the command line, specified as the comma-separated pair consisting of 'Display' and a character vector. 'off' (default) or 'none' displays no output. 'final' displays a message when a computation finishes. 'iter' displays output at each iteration.

Example: 'Display', 'final'

### **UseParallel** — Logical flag to compute confidence intervals in parallel

true | false

Logical flag to compute confidence intervals in parallel, specified as the comma-separated pair consisting of 'UseParallel' and true or false. By default, the parallel options in

the original fit are used. If this argument is set to `true` and Parallel Computing Toolbox is available, the parallel options in the original fit are ignored, and confidence intervals are computed in parallel.

For the Gaussian confidence intervals:

- If the input `fitResults` is a vector of results objects, then the computation of confidence intervals for each object is performed in parallel. The Gaussian confidence intervals are quick to compute. So, it might be more beneficial to parallelize the original fit (`sbiofit`) and not set `UseParallel` to `true` for `sbioparameterci`.

For the Profile Likelihood confidence intervals:

- If the number of results objects in the input `fitResults` vector is greater than the number of estimated parameters, then the computation of confidence intervals for each object is performed in parallel.
- Otherwise, the confidence intervals for all estimated parameters within one results object are computed in parallel before the function moves on to the next results object.

For the Bootstrap confidence intervals:

- The function forwards the `UseParallel` flag to `bootci`. There is no parallelization over the input vector of results objects.

---

**Note** If you have a global stream for random number generation with several substreams to compute in parallel in a reproducible fashion, `sbioparameterci` first checks to see if the number of workers is same as the number of substreams. If so, `sbioparameterci` sets `UseSubstreams` to `true` in the `statset` option and passes it to `bootci`. Otherwise, the substreams are ignored by default.

---

Example: `'UseParallel',true`

## Properties

### Type — Confidence interval type

`'gaussian' | 'profileLikelihood' | 'bootstrap'`

This property is read-only.

Confidence interval type, specified as 'gaussian', 'profileLikelihood', or 'bootstrap'.

Example: 'bootstrap'

### **Alpha — Confidence level**

positive scalar

This property is read-only.

Confidence level,  $(1 - \text{Alpha}) * 100\%$ , specified as a positive scalar between 0 and 1.

Example: 0.01

### **GroupNames — Original group names from data used for fitting**

cell array of character vectors

This property is read-only.

Original group names from the data used for fitting the model, specified as a cell array of character vectors. Each cell contains the name of a group.

Example: {'1'}{'2'}{'3'}

### **Results — Confidence interval results**

table

This property is read-only.

Confidence interval results, specified as a table. The table contains the following columns.

<b>Column Name</b>	<b>Description</b>
<i>Name</i>	Name of the estimated parameter
<i>Estimate</i>	Estimated parameter value
<i>Bounds</i>	Lower and upper parameter bounds (if defined in the original fit)
<i>Group</i>	Group name (if available)
<i>CategoryVariableName</i>	Name of category (if defined in the original fit)

Column Name	Description
<i>CategoryValue</i>	Value of the category variable specified by <i>CategoryVariableName</i>
<i>ConfidenceInterval</i>	Confidence interval values
<i>Status</i>	Confidence interval estimation status, specified as one of the following categorical values: <code>success</code> , <code>constrained</code> , <code>estimable</code> , <code>not estimable</code> (for details, see “Parameter Confidence Interval Estimation Status” on page 2-374)

**ExitFlags** — Exit flags returned during calculation of bootstrap confidence intervals

vector

This property is read-only.

Exit flags returned during the calculation of `bootstrap` confidence intervals only, specified as a vector of integers. Each integer is an exit flag returned by the estimation function (except `nlinfit`) used to fit parameters during bootstrapping. The same estimation function used in the original fit is used for bootstrapping.

Each flag indicates the success or failure status of the fitting performed to create a bootstrap sample. Refer to the reference page of the corresponding estimation function for the meaning of the exit flag.

If the estimation function does not return an exit flag, `ExitFlags` is set to `[]`. For the `gaussian` and `profileLikelihood` confidence intervals, `ExitFlags` is not supported and is always set to `[]`.

**Object Functions**

- `ci2table` Return summary table of confidence interval results
- `plot` Plot parameter confidence interval results

**Examples**

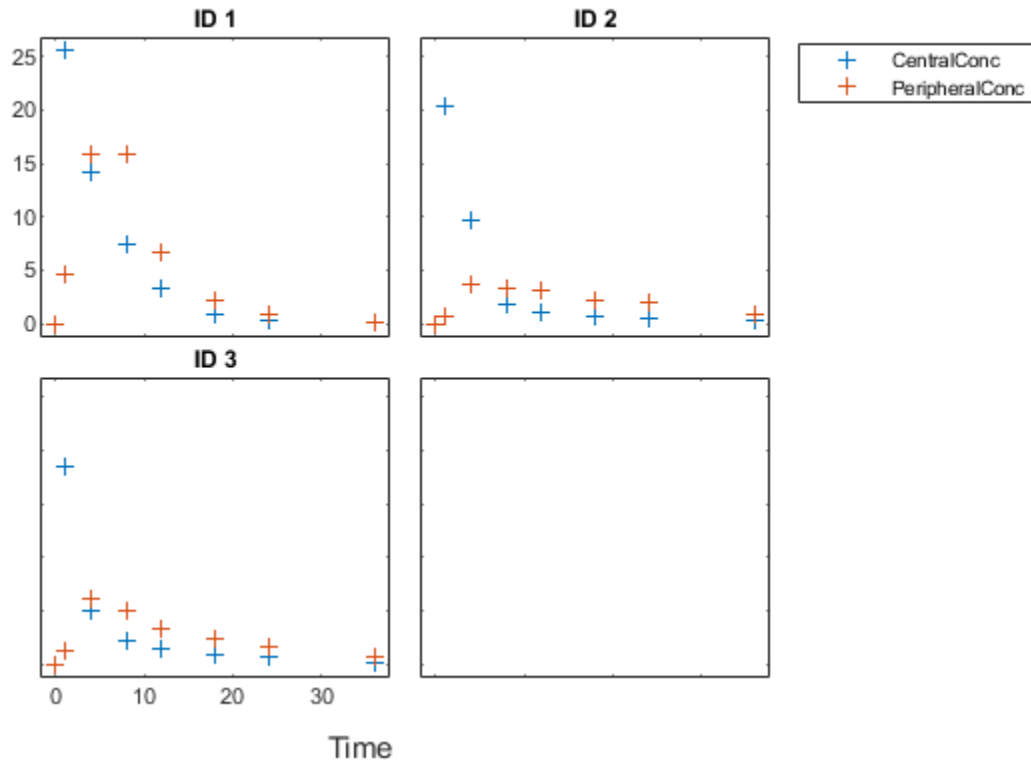


## Compute Confidence Intervals for Estimated PK Parameters and Model Predictions

### Load Data

Load the sample data to fit. The data is stored as a table with variables *ID* , *Time* , *CentralConc* , and *PeripheralConc*. This synthetic data represents the time course of plasma concentrations measured at eight different time points for both central and peripheral compartments after an infusion dose for three individuals.

```
clear all
load data10_32R.mat
gData = groupedData(data);
gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};
sbiotrellis(gData,'ID','Time',{'CentralConc','PeripheralConc'},'Marker','+',...
            'LineStyle','none');
```



### Create Model

Create a two-compartment model.

```
pkmd = PKModelDesign;
pkc1 = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2 = addCompartment(pkmd, 'Peripheral');
model = construct(pkmd);
configset = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

## Define Dosing

Define the infusion dose.

```
dose                = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime     = 0;
dose.Amount        = 100;
dose.Rate          = 50;
dose.AmountUnits   = 'milligram';
dose.TimeUnits     = 'hour';
dose.RateUnits     = 'milligram/hour';
```

## Define Parameters

Define the parameters to estimate. Set the parameter bounds for each parameter. In addition to these explicit bounds, the parameter transformations (such as log, logit, or probit) impose implicit bounds.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam = estimatedInfo(paramsToEstimate, ...
                               'InitialValue', [1 1 1 1], ...
                               'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

## Fit Model

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

Perform a pooled fit, that is, one set of estimated parameters for all patients.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true);
```

## Compute Confidence Intervals for Estimated Parameters

Compute 95% confidence intervals for each estimated parameter in the unpooled fit.

```
ciParamUnpooled = sbioparameterci(unpooledFit);
```

## Display Results

Display the confidence intervals in a table format. For details about the meaning of each estimation status, see “Parameter Confidence Interval Estimation Status” on page 2-374.

```
ci2table(ciParamUnpooled)
```

```
ans =
```

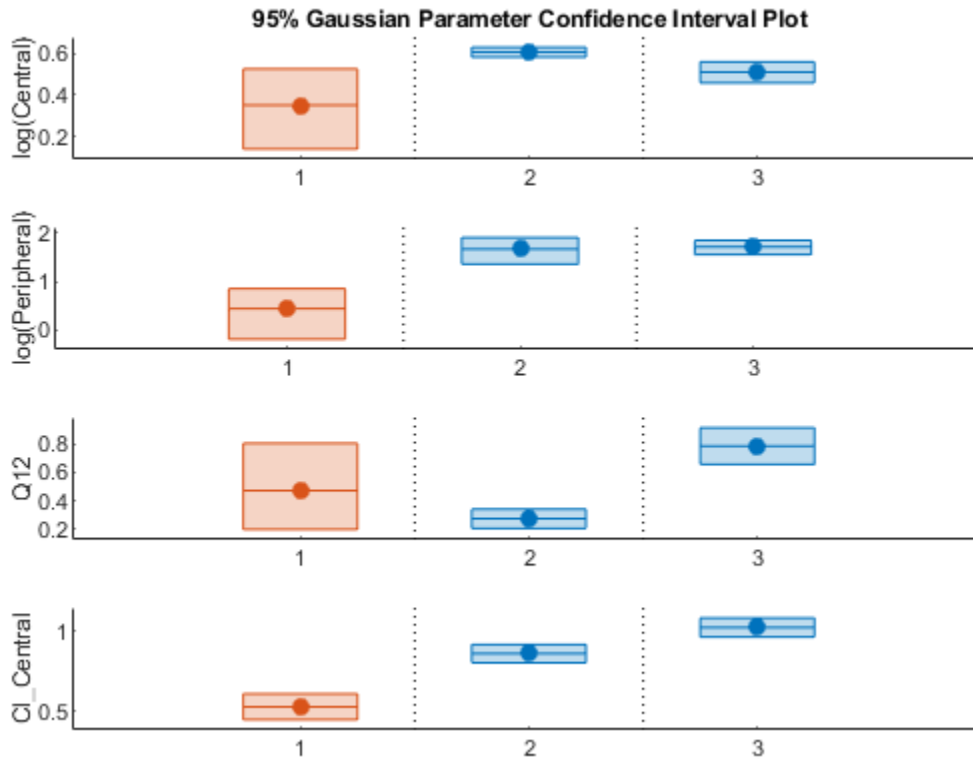
```
12x7 table
```

Group	Name	Estimate	ConfidenceInterval		Type	Alpha	
1	'Central'	1.422	1.1533	1.6906	Gaussian	0.05	est
1	'Peripheral'	1.5629	0.83143	2.3551	Gaussian	0.05	con
1	'Q12'	0.47159	0.20093	0.80247	Gaussian	0.05	con
1	'Cl_Central'	0.52898	0.44842	0.60955	Gaussian	0.05	est
2	'Central'	1.8322	1.7893	1.8751	Gaussian	0.05	su
2	'Peripheral'	5.3368	3.9133	6.7602	Gaussian	0.05	su
2	'Q12'	0.27641	0.2093	0.34351	Gaussian	0.05	su
2	'Cl_Central'	0.86034	0.80313	0.91755	Gaussian	0.05	su
3	'Central'	1.6657	1.5818	1.7497	Gaussian	0.05	su
3	'Peripheral'	5.5632	4.7557	6.3708	Gaussian	0.05	su
3	'Q12'	0.78361	0.65581	0.91142	Gaussian	0.05	su
3	'Cl_Central'	1.0233	0.96375	1.0828	Gaussian	0.05	su

Plot the confidence intervals. If the estimation status of a confidence interval is **success**, it is plotted in blue (the first default color). Otherwise, it is plotted in red (the second default color), which indicates that further investigation into the fitted parameters may be required. If the confidence interval is **not estimable**, then the function plots a red line with a centered cross. If there are any transformed parameters with estimated values 0 (for the log transform) and 1 or 0 (for the probit or logit transform), then no confidence intervals are plotted for those parameter estimates. To see the color order, type `get(groot, 'defaultAxesColorOrder')`.

Groups are displayed from left to right in the same order that they appear in the `GroupNames` property of the object, which is used to label the x-axis. The y-labels are the transformed parameter names.

```
plot(ciParamUnpooled)
```



Compute the confidence intervals for the pooled fit.

```
ciParamPooled = sbioparameterci(pooledFit);
```

Display the confidence intervals.

```
ci2table(ciParamPooled)
```

ans =

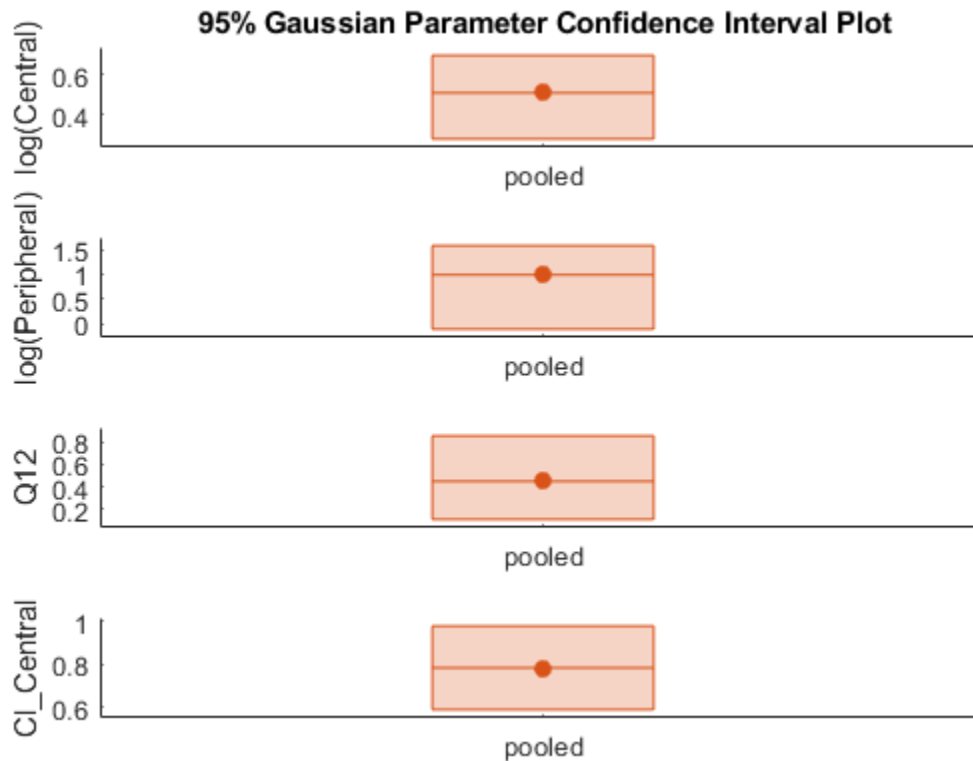
4x7 table

Group	Name	Estimate	ConfidenceInterval	Type	Alpha
-------	------	----------	--------------------	------	-------

pooled	'Central'	1.6626	1.3287	1.9965	Gaussian	0.05	es
pooled	'Peripheral'	2.687	0.89848	4.8323	Gaussian	0.05	co
pooled	'Q12'	0.44956	0.11445	0.85152	Gaussian	0.05	co
pooled	'Cl_Central'	0.78493	0.59222	0.97764	Gaussian	0.05	es

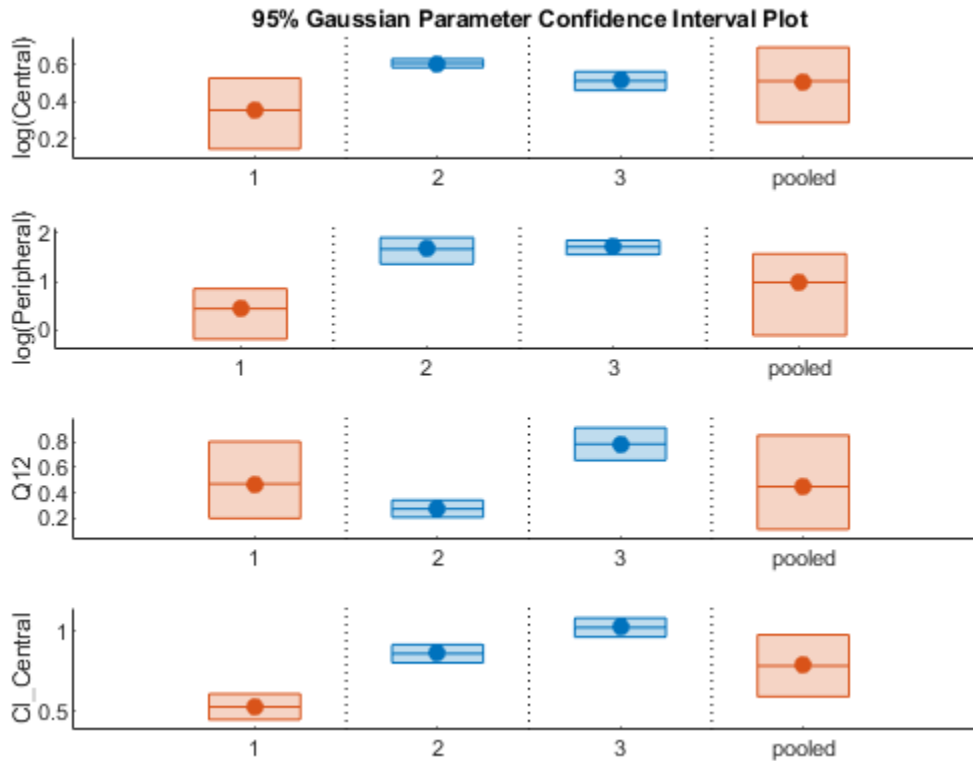
Plot the confidence intervals. The group name is labeled as "pooled" to indicate such fit.

```
plot(ciParamPooled)
```



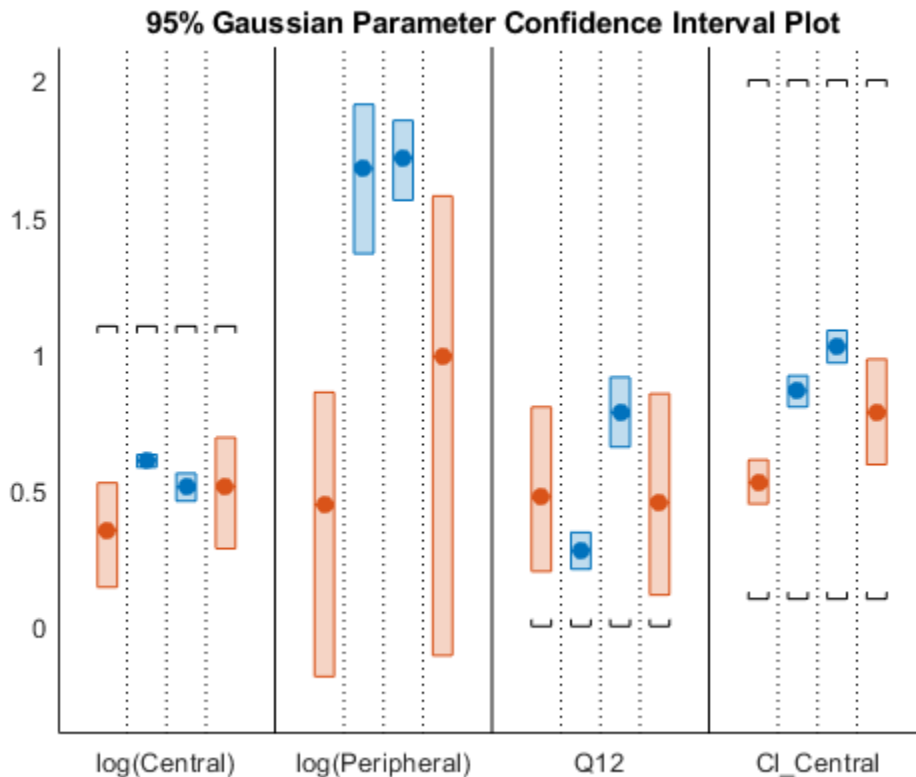
Plot all the confidence interval results together. By default, the confidence interval for each parameter estimate is plotted on a separate axes. Vertical lines group confidence intervals of parameter estimates that were computed in a common fit.

```
ciAll = [ciParamUnpooled;ciParamPooled];
plot(ciAll)
```



You can also plot all confidence intervals in one axes grouped by parameter estimates using the 'Grouped' layout.

```
plot(ciAll, 'Layout', 'Grouped')
```



In this layout, you can point to the center marker of each confidence interval to see the group name. Each estimated parameter is separated by a vertical black line. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets. Note the different scales on the y-axis due to parameter transformations. For instance, the y-axis of Q12 is in the linear scale, but that of Central is in the log scale due to its log transform.

### Compute Confidence Intervals for Model Predictions

Calculate 95% confidence intervals for the model predictions, that is, simulation results using the estimated parameters.



```

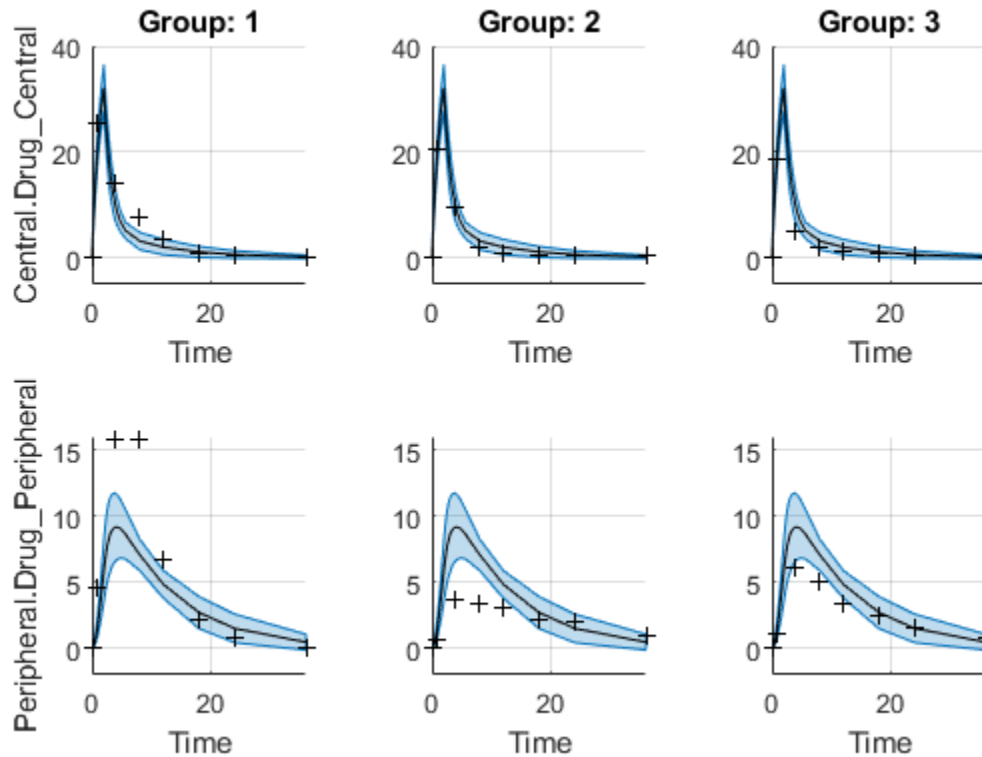
% For the pooled fit
ciPredPooled = sbiopredictionci(pooledFit);
% For the unpooled fit
ciPredUnpooled = sbiopredictionci(unpooledFit);

```

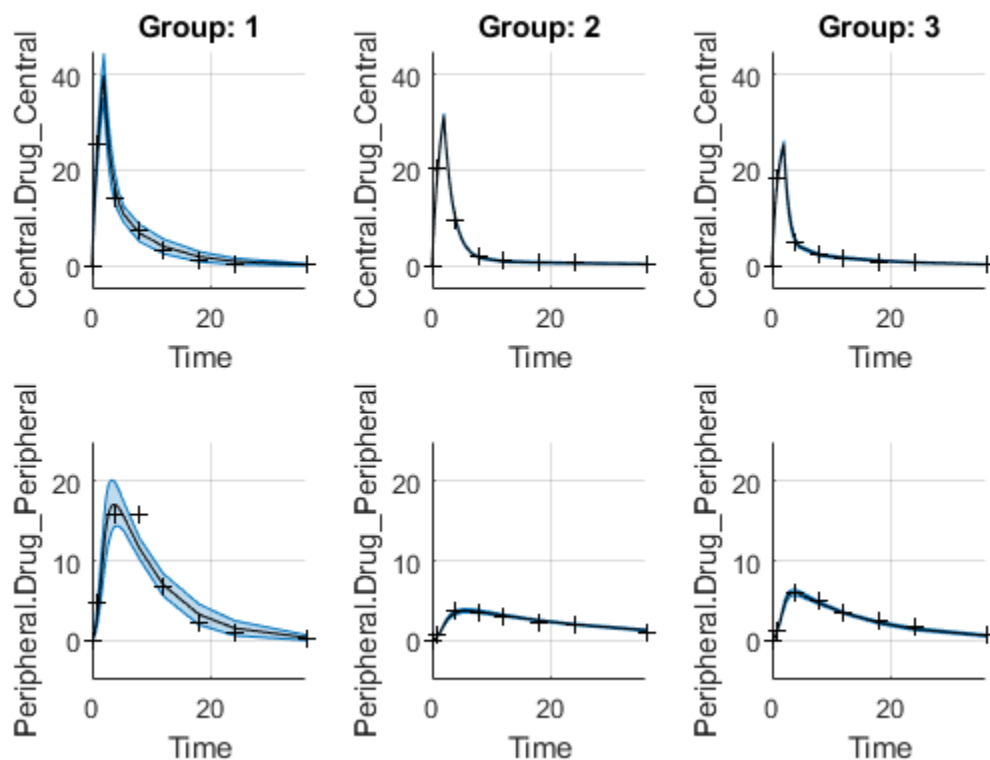
### Plot Confidence Intervals for Model Predictions

The confidence interval for each group is plotted in a separate column, and each response is plotted in a separate row. Confidence intervals limited by the bounds are plotted in red. Confidence intervals not limited by the bounds are plotted in blue.

```
plot(ciPredPooled)
```



```
plot(ciPredUnpooled)
```



## Definitions

### Parameter Confidence Interval Estimation Status

The following are the definitions of confidence interval estimation statuses for different types of confidence intervals.

- `not estimable` - The confidence interval is unbounded.
- `constrained` - The confidence interval is constrained by a parameter bound defined in the original fit. Parameter transformations (such as `log`, `probit`, or `logit`) impose

implicit bounds on the estimated parameters, for example, positivity constraints. Such bounds can lead to the overestimation of the confidence, that is, the confidence interval can be smaller than expected.

- **success** - All confidence intervals for all parameters are computed successfully.
- **estimable** - The confidence interval is computed successfully, but other parameters have an estimation status of **not estimable** or **constrained**.

For more details about the algorithm, see “Gaussian Confidence Interval Calculation” on page 1-209.

- **not estimable** - The computation of the confidence interval is unsuccessful. This can happen when the profile likelihood curve is not strictly monotonically decreasing, or due to computation failures in the profile likelihood.
- **constrained** - The profile likelihood curve is bounded by the bounds on the estimated parameters defined in the original fit. Parameter transformations, such as **log**, **logit**, **probit**, impose implicit bounds on the estimated parameters, for example, positivity constraints.
- **success** - If there is no parameter estimate with the confidence interval estimation status **constrained** or **not estimable**, then the function sets all estimation statuses to **success**.
- **estimable** - The confidence interval is computed successfully, but other parameters have an estimation status of **not estimable** or **constrained**.

For more details about the algorithm, see “Profile Likelihood Confidence Interval Calculation” on page 1-210.

- **constrained** - The confidence interval is closer than **Tolerance** to the parameter bounds defined in the original fit.
- **success** - All confidence intervals were further away from the parameter bounds than **Tolerance**.
- **estimable** - The confidence interval is computed successfully, but other parameters have an estimation status of **constrained**.

For more details about the algorithm, see “Bootstrap Confidence Interval Calculation” on page 1-212.

## **See Also**

PredictionConfidenceInterval | sbioparameterci | sbiopredictionci

**Introduced in R2017b**

# PredictionConfidenceInterval

Object containing confidence interval results for model predictions

## Description

The `PredictionConfidenceInterval` object contains confidence interval results for model predictions (that is, simulation results based on estimated parameters), computed using `sbiopredictionci`.

## Creation

## Syntax

```
ci = sbiopredictionci(fitResults)
ci = sbiopredictionci(fitResults,Name,Value)
```

## Description

`ci = sbiopredictionci(fitResults)` computes 95% confidence intervals for the model simulation results from `fitResults`, a `NLINResults` object or `OptimResults` object returned by `sbiofit`. `ci` is a `PredictionConfidenceInterval` object that contains the computed confidence interval data.

`ci = sbiopredictionci(fitResults,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

## Input Arguments

**fitResults** — Parameter estimation results from `sbiofit`

`NLINResults` object | `OptimResults` object | vector

Parameter estimation results from `sbiofit`, specified as an `NLINResults` object, `OptimResults` object, or a vector of objects for unpooled fits that were returned from the same `sbiofit` call.

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `'Alpha', 0.01, 'Type', 'bootstrap'` specifies to compute a 99% confidence interval using the bootstrap method.

### **Alpha — Confidence level**

0.05 (default) | positive scalar

Confidence level,  $(1 - \textit{Alpha}) * 100\%$ , specified as the comma-separated pair consisting of `'Alpha'` and a positive scalar between 0 and 1. The default value is 0.05, meaning a 95% confidence interval is computed.

Example: `'Alpha', 0.01`

### **Type — Confidence interval type**

`'gaussian'` (default) | `'bootstrap'`

Confidence interval type, specified as the comma-separated pair consisting of `'Type'` and a character vector. The valid choices are:

- `'gaussian'` - Use the Gaussian approximation on page 1-238 of the distribution of the linearized model responses around the parameter estimates.
- `'bootstrap'` - Compute confidence intervals using the bootstrap method on page 1-239.

Example: `'Type', 'bootstrap'`

### **NumSamples — Number of samples for bootstrapping**

1000 (default) | positive integer

Number of samples for bootstrapping, specified as the comma-separated pair consisting of `'NumSamples'` and a positive integer. This number defines the number of fits that are performed during the confidence interval computation to generate bootstrap samples. The smaller the number is, the faster the computation of the confidence intervals becomes, at the cost of decreased accuracy.

Example: 'NumSamples',500

**Display – Level of display returned to the command line**

'off' (default) | 'none' | 'final'

Level of display returned to the command line, specified as the comma-separated pair consisting of 'Display' and a character vector. 'off' (default) or 'none' displays no output. 'final' displays a message when the computation finishes.

Example: 'Display','final'

**UseParallel – Logical flag to compute confidence intervals in parallel**

true | false

Logical flag to compute confidence intervals in parallel, specified as the comma-separated pair consisting of 'UseParallel' and true or false. By default, the parallel options in the original fit are used. If this argument is set to true and Parallel Computing Toolbox is available, the parallel options in the original fit are ignored, and confidence intervals are computed in parallel.

For the Gaussian confidence intervals:

- If the input `fitResults` is a vector of results objects, then the computation of confidence intervals for each object is performed in parallel. The Gaussian confidence intervals are quick to compute. So, it might be more beneficial to parallelize the original fit (`sbiofit`) and not set `UseParallel` to true for `sbiopredictionci`.

For the Bootstrap confidence intervals:

- The function forwards the `UseParallel` flag to `bootci`. There is no parallelization over the input vector of results objects.

---

**Note** If you have a global stream for random number generation with a number of substreams to compute in parallel in a reproducible fashion, `sbiopredictionci` first checks to see if the number of workers is same as the number of substreams. If so, the function sets `UseSubstreams` to true in the `statset` option and passes to `bootci`. Otherwise, the substreams are ignored by default.

---

Example: 'UseParallel',true

## Properties

### **ResponseNames — Names of model responses**

cell array of character vectors

This property is read-only.

Names of model responses in the parameter fit, specified as a cell array of character vectors. Each cell contains the name of a response.

Example: {'Central.Drug\_Central' } {'Peripheral.Drug\_Peripheral'}

### **Status — Confidence interval estimation status**

categorical

This property is read-only.

Confidence interval estimation status, specified as one of the following categorical values:

- **success** - The proper confidence intervals are found. That is, no model prediction is constrained by the parameter bounds defined in the original fit.
- **constrained** - The confidence intervals are found, but the confidence interval for the model response is constrained by a parameter bound defined in the original fit.
- **not estimable** - No confidence intervals are found.
- **estimable** - The proper confidence intervals are found, but other model predictions have an estimation status of either **constrained** or **not estimable**. For the bootstrap confidence interval, the status is always set to **estimable**.

For details, see “Gaussian Confidence Interval Calculation for Model Predictions” on page 1-238 and “Bootstrap Confidence Interval Calculation” on page 1-239.

Example: `success`

### **Type — Confidence interval type**

'gaussian' | 'bootstrap'

This property is read-only.

Confidence interval type, specified as 'gaussian' or 'bootstrap'.

Example: 'bootstrap'



**Alpha — Confidence level**

positive scalar

This property is read-only.

Confidence level,  $(1 - \text{Alpha}) * 100\%$ , specified as a positive scalar between 0 and 1.

Example: 0.01

**GroupNames — Original group names from data used for fitting**

cell array of character vectors

This property is read-only.

Original group names from the data used for fitting the model, specified as a cell array of character vectors. Each cell contains the name of a group.

Example: {'1'}{'2'}{'3'}

**Results — Confidence interval results**

table

This property is read-only.

Confidence interval results, specified as a table. The table contains the following columns.

Column Name	Description
<i>Group</i>	Group name
<i>Response</i>	Model response name
<i>Time</i>	Simulation time
<i>Estimate</i>	Estimated response value
<i>ConfidenceInterval</i>	Confidence interval values

**ExitFlags — Exit flags returned during calculation of bootstrap confidence intervals**

vector

This property is read-only.

Exit flags returned during the calculation of `bootstrap` confidence intervals only, specified as a vector of integers. Each integer is an exit flag returned by the estimation

function (except `nlinfit`) used to fit parameters during bootstrapping. The same estimation function used in the original fit is used for bootstrapping.

Each flag indicates the success or failure status of the fitting performed to create a bootstrap sample. Refer to the reference page of the corresponding estimation function for the meaning of the exit flag.

If the estimation function does not return an exit flag, `ExitFlags` is set to `[]`. For the gaussian confidence intervals, `ExitFlags` is not supported and is always set to `[]`.

## Object Functions

`plot` Plot confidence interval results for model predictions

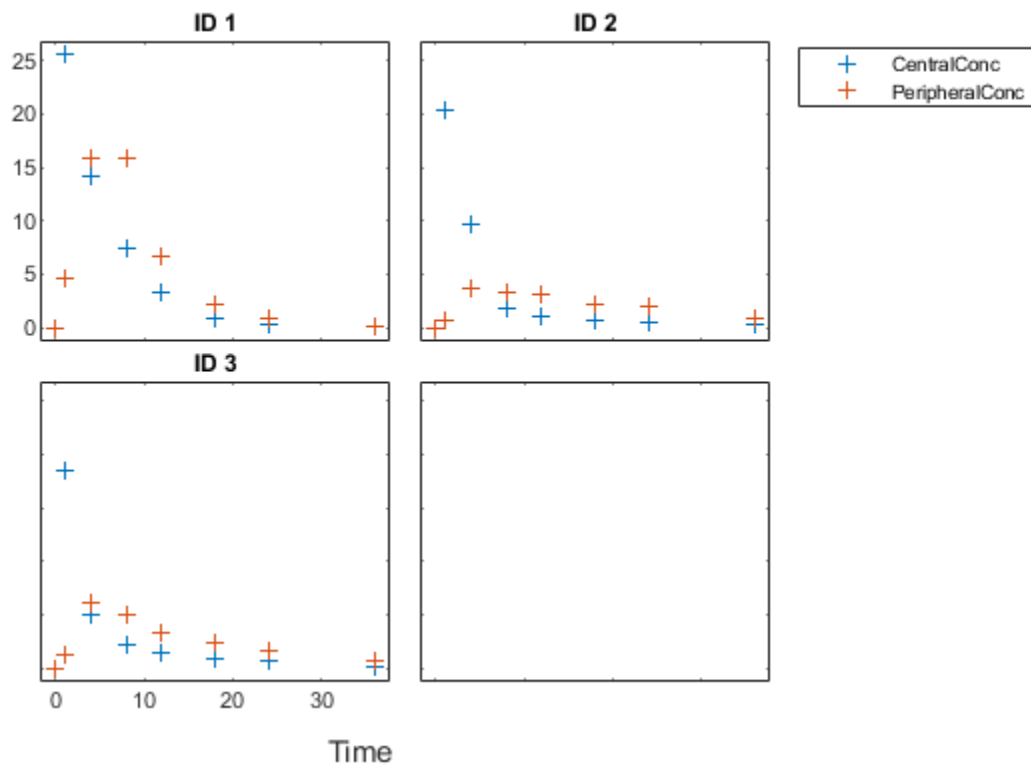
## Examples

### Compute Confidence Intervals for Estimated PK Parameters and Model Predictions

#### Load Data

Load the sample data to fit. The data is stored as a table with variables *ID*, *Time*, *CentralConc*, and *PeripheralConc*. This synthetic data represents the time course of plasma concentrations measured at eight different time points for both central and peripheral compartments after an infusion dose for three individuals.

```
clear all
load data10_32R.mat
gData = groupedData(data);
gData.Properties.VariableUnits = {'','hour','milligram/liter','milligram/liter'};
sbiotrellis(gData,'ID','Time',{'CentralConc','PeripheralConc'},'Marker','+',...
            'LineStyle','none');
```



### Create Model

Create a two-compartment model.

```
pkmd          = PKModelDesign;
pkc1         = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Infusion';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
pkc2         = addCompartment(pkmd, 'Peripheral');
model        = construct(pkmd);
configset    = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

### **Define Dosing**

Define the infusion dose.

```
dose                = sbiodose('dose', 'TargetName', 'Drug_Central');
dose.StartTime     = 0;
dose.Amount        = 100;
dose.Rate          = 50;
dose.AmountUnits   = 'milligram';
dose.TimeUnits     = 'hour';
dose.RateUnits     = 'milligram/hour';
```

### **Define Parameters**

Define the parameters to estimate. Set the parameter bounds for each parameter. In addition to these explicit bounds, the parameter transformations (such as log, logit, or probit) impose implicit bounds.

```
responseMap = {'Drug_Central = CentralConc', 'Drug_Peripheral = PeripheralConc'};
paramsToEstimate = {'log(Central)', 'log(Peripheral)', 'Q12', 'Cl_Central'};
estimatedParam = estimatedInfo(paramsToEstimate, ...
                               'InitialValue', [1 1 1 1], ...
                               'Bounds', [0.1 3; 0.1 10; 0 10; 0.1 2]);
```

### **Fit Model**

Perform an unpooled fit, that is, one set of estimated parameters for each patient.

```
unpooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', false);
```

Perform a pooled fit, that is, one set of estimated parameters for all patients.

```
pooledFit = sbiofit(model, gData, responseMap, estimatedParam, dose, 'Pooled', true);
```

### **Compute Confidence Intervals for Estimated Parameters**

Compute 95% confidence intervals for each estimated parameter in the unpooled fit.

```
ciParamUnpooled = sbioparameterci(unpooledFit);
```

### **Display Results**

Display the confidence intervals in a table format. For details about the meaning of each estimation status, see “Parameter Confidence Interval Estimation Status” on page 2-374.

```
ci2table(ciParamUnpooled)
```

```
ans =
```

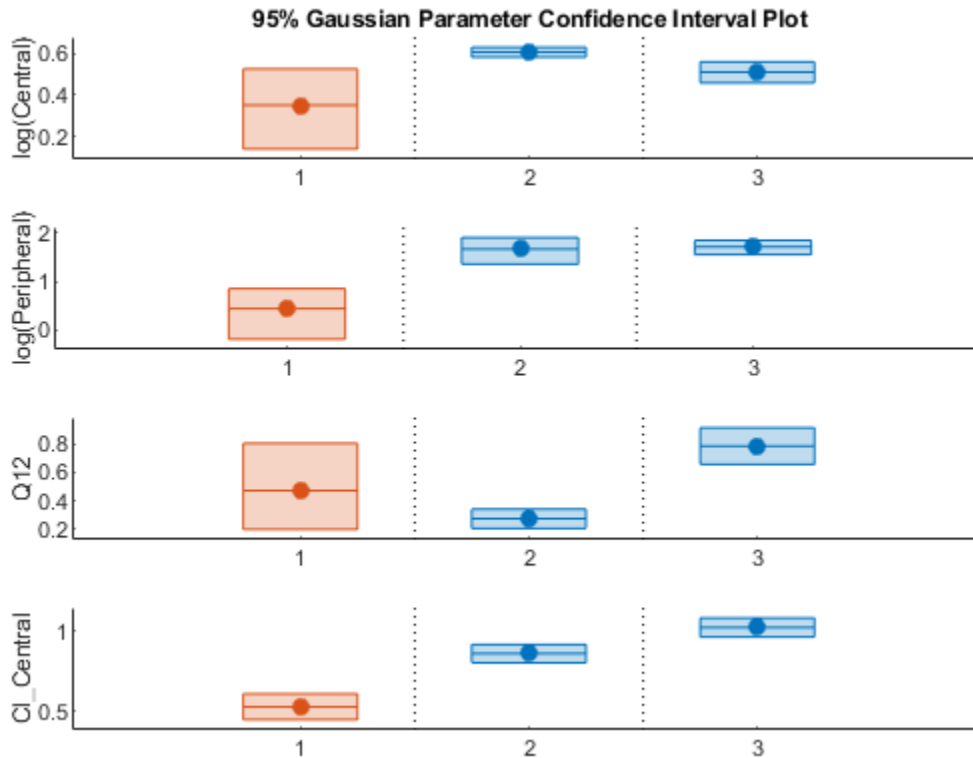
```
12x7 table
```

Group	Name	Estimate	ConfidenceInterval		Type	Alpha	
1	'Central'	1.422	1.1533	1.6906	Gaussian	0.05	est
1	'Peripheral'	1.5629	0.83143	2.3551	Gaussian	0.05	con
1	'Q12'	0.47159	0.20093	0.80247	Gaussian	0.05	con
1	'Cl_Central'	0.52898	0.44842	0.60955	Gaussian	0.05	est
2	'Central'	1.8322	1.7893	1.8751	Gaussian	0.05	su
2	'Peripheral'	5.3368	3.9133	6.7602	Gaussian	0.05	su
2	'Q12'	0.27641	0.2093	0.34351	Gaussian	0.05	su
2	'Cl_Central'	0.86034	0.80313	0.91755	Gaussian	0.05	su
3	'Central'	1.6657	1.5818	1.7497	Gaussian	0.05	su
3	'Peripheral'	5.5632	4.7557	6.3708	Gaussian	0.05	su
3	'Q12'	0.78361	0.65581	0.91142	Gaussian	0.05	su
3	'Cl_Central'	1.0233	0.96375	1.0828	Gaussian	0.05	su

Plot the confidence intervals. If the estimation status of a confidence interval is **success**, it is plotted in blue (the first default color). Otherwise, it is plotted in red (the second default color), which indicates that further investigation into the fitted parameters may be required. If the confidence interval is **not estimable**, then the function plots a red line with a centered cross. If there are any transformed parameters with estimated values 0 (for the log transform) and 1 or 0 (for the probit or logit transform), then no confidence intervals are plotted for those parameter estimates. To see the color order, type `get(groot, 'defaultAxesColorOrder')`.

Groups are displayed from left to right in the same order that they appear in the `GroupNames` property of the object, which is used to label the x-axis. The y-labels are the transformed parameter names.

```
plot(ciParamUnpooled)
```



Compute the confidence intervals for the pooled fit.

```
ciParamPooled = sbioparameterci(pooledFit);
```

Display the confidence intervals.

```
ci2table(ciParamPooled)
```

```
ans =
```

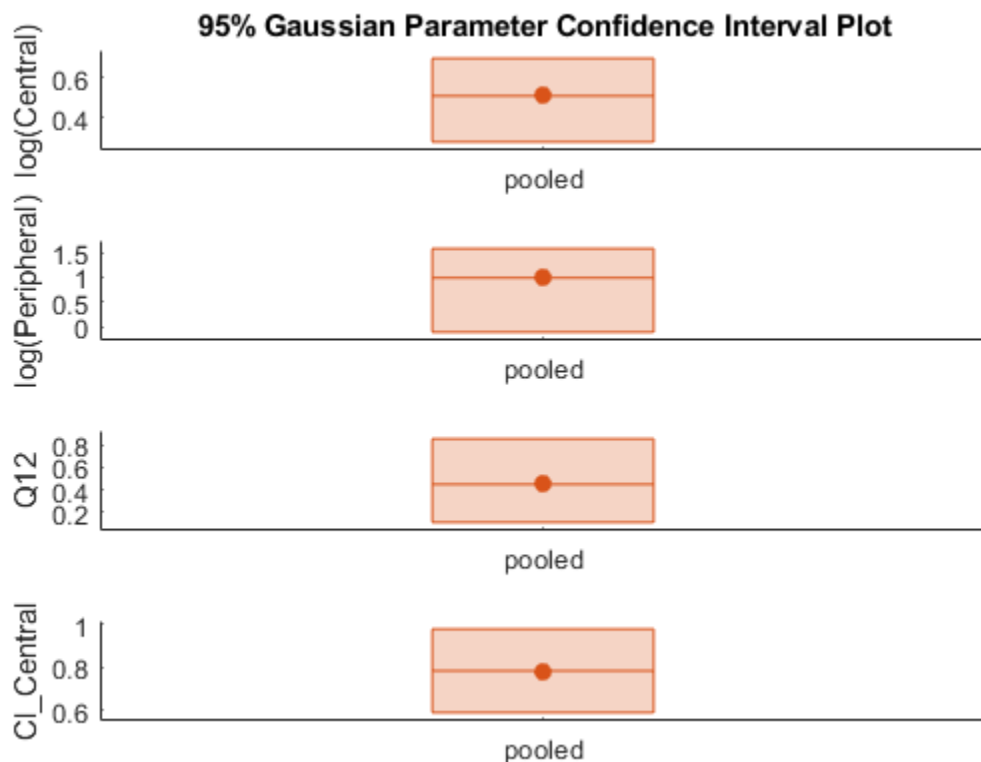
```
4x7 table
```

Group	Name	Estimate	ConfidenceInterval	Type	Alpha
-------	------	----------	--------------------	------	-------

pooled	'Central'	1.6626	1.3287	1.9965	Gaussian	0.05	es
pooled	'Peripheral'	2.687	0.89848	4.8323	Gaussian	0.05	co
pooled	'Q12'	0.44956	0.11445	0.85152	Gaussian	0.05	co
pooled	'Cl_Central'	0.78493	0.59222	0.97764	Gaussian	0.05	es

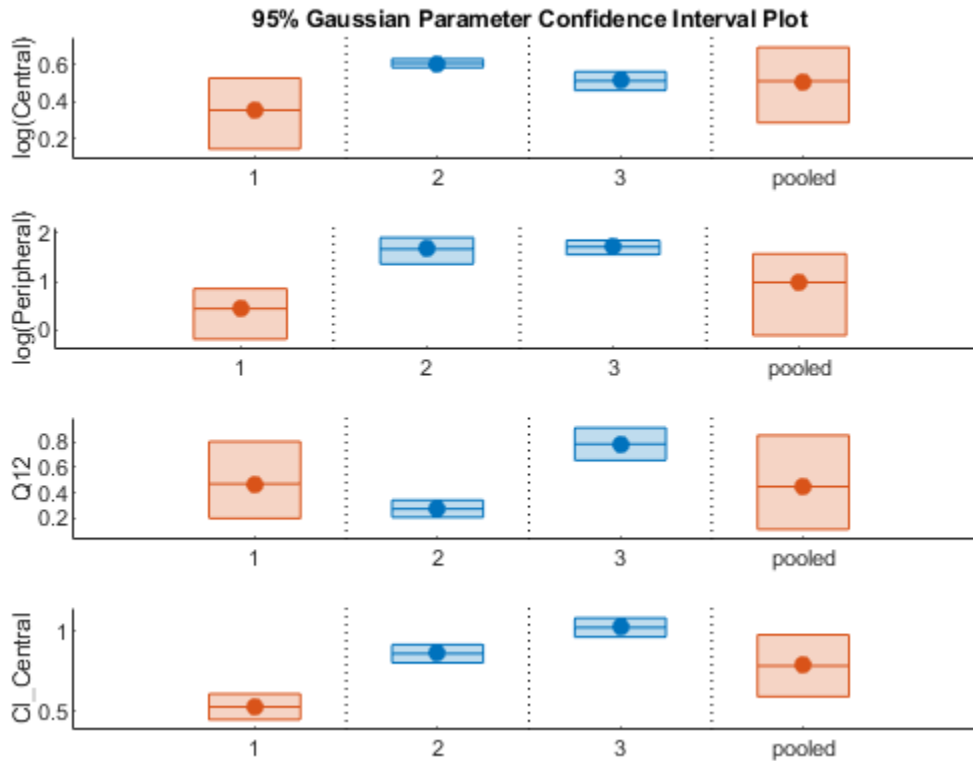
Plot the confidence intervals. The group name is labeled as "pooled" to indicate such fit.

```
plot(ciParamPooled)
```



Plot all the confidence interval results together. By default, the confidence interval for each parameter estimate is plotted on a separate axes. Vertical lines group confidence intervals of parameter estimates that were computed in a common fit.

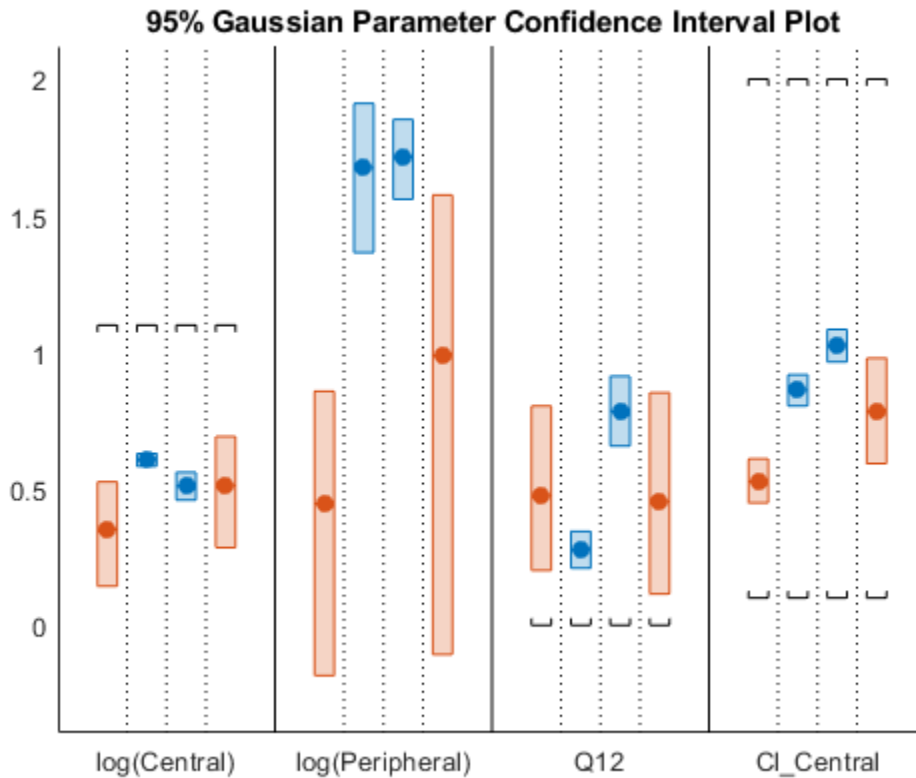
```
ciAll = [ciParamUnpooled;ciParamPooled];
plot(ciAll)
```



You can also plot all confidence intervals in one axes grouped by parameter estimates using the 'Grouped' layout.

```
plot(ciAll, 'Layout', 'Grouped')
```





In this layout, you can point to the center marker of each confidence interval to see the group name. Each estimated parameter is separated by a vertical black line. Vertical dotted lines group confidence intervals of parameter estimates that were computed in a common fit. Parameter bounds defined in the original fit are marked by square brackets. Note the different scales on the y-axis due to parameter transformations. For instance, the y-axis of Q12 is in the linear scale, but that of Central is in the log scale due to its log transform.

### Compute Confidence Intervals for Model Predictions

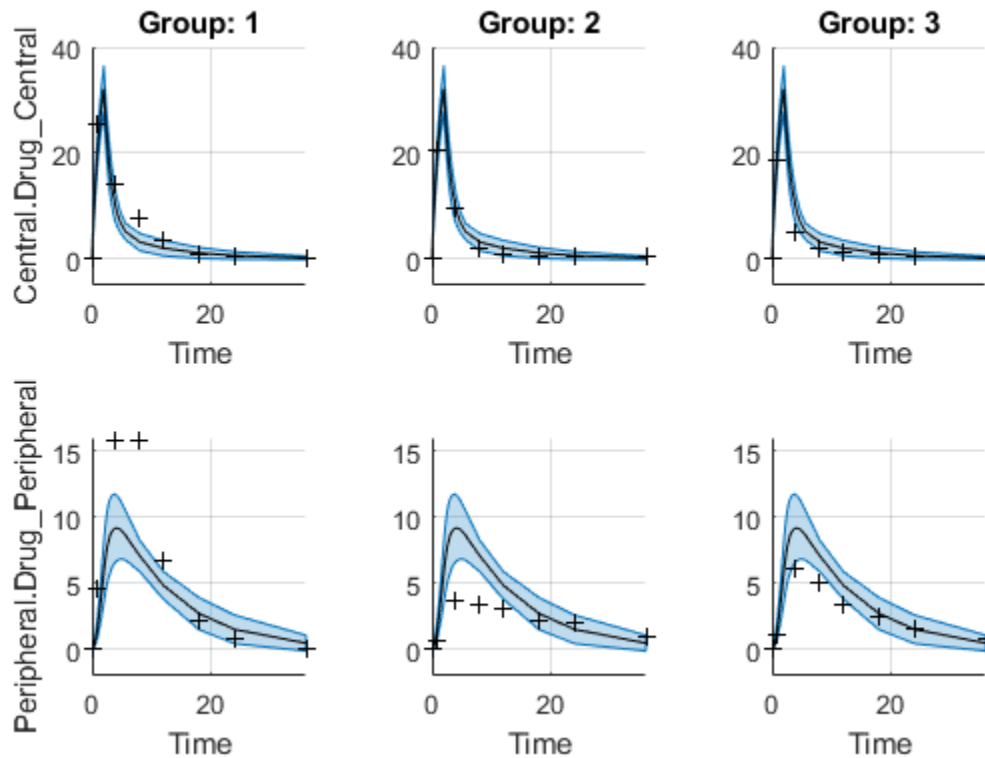
Calculate 95% confidence intervals for the model predictions, that is, simulation results using the estimated parameters.

```
% For the pooled fit
ciPredPooled = sbiopredictionci(pooledFit);
% For the unpooled fit
ciPredUnpooled = sbiopredictionci(unpooledFit);
```

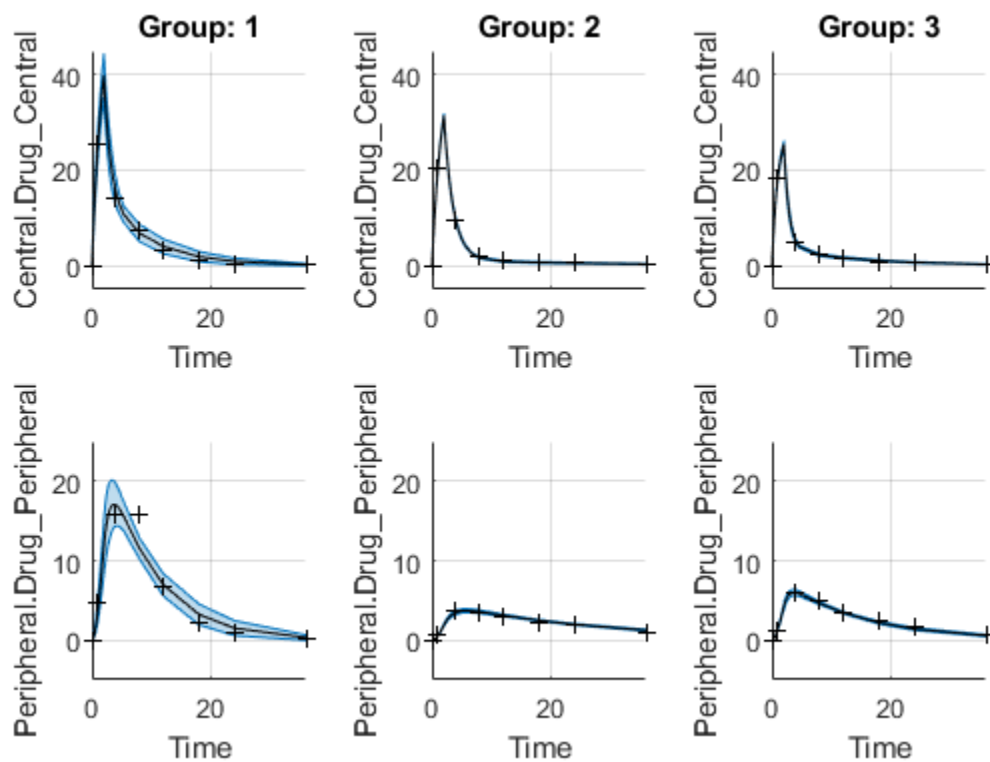
### Plot Confidence Intervals for Model Predictions

The confidence interval for each group is plotted in a separate column, and each response is plotted in a separate row. Confidence intervals limited by the bounds are plotted in red. Confidence intervals not limited by the bounds are plotted in blue.

```
plot(ciPredPooled)
```



```
plot(ciPredUnpooled)
```



## See Also

[ParameterConfidenceInterval](#) | [sbioparameterci](#) | [sbiopredictionci](#)

Introduced in R2017b

# Reaction object

Options for model reactions

## Description

The reaction object represents a *reaction*, which describes a transformation, transport, or binding process that changes one or more species. Typically, the change is the amount of a species. For example:

```
Creatine + ATP <-> ADP + phosphocreatine
```

```
glucose + 2 ADP + 2 Pi -> 2 lactic acid + 2 ATP + 2 H2O
```

Spaces are required before and after species names and stoichiometric values.

See “Property Summary” on page 2-394 for links to reaction object property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the graphical user interface.

## Constructor Summary

`addreaction (model)`      Create reaction object and add to model object

## Method Summary

addkineticlaw (reaction)	Create kinetic law object and add to reaction object
addproduct (reaction)	Add product species object to reaction object
addreactant (reaction)	Add species object as reactant to reaction object
copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
get (any object)	Get object properties
rename (compartment, parameter, species, reaction)	Rename object and update expressions
rmproduct (reaction)	Remove species object from reaction object products
rmreactant (reaction)	Remove species object from reaction object reactants
set (any object)	Set object properties

## Property Summary

Active	Indicate object in use during simulation
KineticLaw	Show kinetic law used for ReactionRate
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Products	Array of reaction products
Reactants	Array of reaction reactants
Reaction	Reaction object reaction
ReactionRate	Reaction rate equation in reaction object
Reversible	Specify whether reaction is reversible or irreversible
Stoichiometry	Species coefficients in reaction
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

“Definitions and Evaluations of Reactions”, AbstractKineticLaw object, Configset object, KineticLaw object, Model object, Parameter object, Root object, Rule object, Species object

**Introduced in R2006b**

## removeconfigset (model)

Remove configuration set from model

### Syntax

```
removeconfigset(modelObj, 'NameValue')
removeconfigset(modelObj, configsetObj)
```

### Arguments

<i>modelObj</i>	Model object from which to remove the configuration set.
<i>NameValue</i>	Name of the configuration set.
<i>configsetObj</i>	Configset object that is to be removed from the model object.

### Description

`removeconfigset(modelObj, 'NameValue')` removes the configset object with the name *NameValue* from the SimBiology model object *modelObj*. A configuration set object stores simulation-specific information. A SimBiology model can contain multiple configuration sets with one being active at any given time. The active configuration set contains the settings that are used during the simulation. *modelObj* always contains at least one configuration set object with name configured to 'default'. You cannot remove the default configuration set from *modelObj*. If the active configuration set is removed from *modelObj*, then the default configuration set will be made active.

`removeconfigset(modelObj, configsetObj)` removes the configuration set object, *configsetObj*, from the SimBiology model, *modelObj*. The configuration set is not deleted; if you want to delete *configsetObj*, use the `delete` on page 2-137 method.

If however, there is no MATLAB variable holding the configset, `removeconfigset(modelObj, 'NameValue')` removes the configset from the model and deletes it.

## Examples

- 1 Create a model object by importing the file `oscillator.xml` and add a configset.

```
modelObj = sbmlimport('oscillator');  
configsetObj = addconfigset(modelObj, 'myset');
```

- 2 Remove the configset from `modelObj` by name or alternatively by indexing.

```
% Remove the configset with name 'myset'.  
removeconfigset(modelObj, 'myset');
```

```
% Get all configset objects and remove the second.  
configsetObj = getconfigset(modelObj);  
removeconfigset(modelObj, configsetObj(2));
```

## See Also

Model object, Configset object, `addconfigset`, `getconfigset`, `setactiveconfigset`

**Introduced in R2006a**



## removedose (model)

Add dose object to model

### Syntax

```
doseObj2 = removedose(modelObj, 'DoseName')
doseObj2 = removedose(modelObj, doseObj)
```

### Arguments

<i>modelObj</i>	Model object from which you remove a dose object.
<i>DoseName</i>	Name of the dose object to remove from a model object. <i>DoseName</i> is the value of the dose object property Name.
<i>doseObj</i>	Dose object to remove from a model object.

### Outputs

<i>doseObj2</i>	ScheduleDose or RepeatDose object.
-----------------	------------------------------------

### Description

`doseObj2 = removedose(modelObj, 'DoseName')` removes a SimBiology ScheduleDose or RepeatDose object with the name *DoseName* from a model object (*modelObj*). returns the dose object (*doseObj*), and assigns [] to the dose object property Parent.

You can add a removed dose object back to a model object using the method `adddose`.

`doseObj2 = removedose(modelObj, doseObj)` removes a SimBiology ScheduleDose or RepeatDose object *doseObj*.

### Examples

Remove a dose object from a model object.

- 1 Create model and dose objects, and then add dose to model.

```
modelObj = sbiomodel('mymodel');  
dose1Obj = adddose(modelObj, 'dose1');
```

- 2 Remove dose object from model object.

```
removedose(mymodel, 'dose1');
```

Get all dose objects from a model object, and then remove the second dose object.

```
AllDoseObjects = getdose(mymodel);  
removedose(mymodel, AllDoseObjects(2));
```

### See Also

Model methods:

- `adddose` — add a dose object to a model object
- `getdose` — get dose information from a model object
- `removedose` — remove a dose object from a model object

Dose object constructor `sbiodose`.

`ScheduleDose` object and `RepeatDose` object methods:

- `copyobj` — copy a dose object from one model object to another model object
- `get` — view properties for a dose object
- `set` — define or modify properties for a dose object

**Introduced in R2010a**

## removevariant (model)

Remove variant from model

### Syntax

```
variantObj = removevariant(modelObj, 'NameValue')
```

```
variantObj = removevariant(modelObj, variantObj)
```

### Arguments

<i>modelObj</i>	Specify the <code>model</code> object from which you want to remove the variant.
<i>variantObj</i>	Specify the <code>variant</code> object to return from the model object.

### Description

*variantObj* = removevariant(*modelObj*, 'NameValue') removes a SimBiology variant object with the name *NameValue* from the model object *modelObj* and returns the variant object to *variantObj*. The variant object `Parent` property is assigned [] (empty).

A SimBiology variant object stores alternate values for properties on a SimBiology model. For more information on variants, see `Variant` object.

*variantObj* = removevariant(*modelObj*, *variantObj*) removes a SimBiology variant object (*variantObj*) and returns the variant object *variantObj*.

To view the variants stored on a model object, use the `getvariant` method. To copy a variant object to another model, use `copyobj`. To add a variant object to a SimBiology model, use the `addvariant` method.

## Examples

- 1 Create a model containing several variants.

```
modelObj = sbiomodel('mymodel');  
variantObj1 = addvariant(modelObj, 'v1');  
variantObj2 = addvariant(modelObj, 'v2');  
variantObj3 = addvariant(modelObj, 'v3');
```

- 2 Remove a variant object using its name.

```
removevariant(modelObj, 'v1');
```

- 3 Remove a variant object using its index number.

- a Get the index number of the variant in the model.

```
v0bjs = getvariant(modelObj)
```

```
SimBiology Variant Array
```

Index:	Name:	Active:
1	v2	false
2	v3	false

- b Remove the variant object.

```
removevariant(modelObj, v0bjs(2));
```

## See Also

addvariant, getvariant, model object, variant object

**Introduced in R2007b**

## rename (compartment, parameter, species, reaction)

Rename object and update expressions

### Syntax

```
rename(Obj, 'NewNameValue')
```

### Arguments

<i>Obj</i>	Compartment, parameter, species, or reaction object.
'NewNameValue'	Specify the new name.

### Description

`rename(Obj, 'NewNameValue')`, changes the `Name` property of the object, *Obj* to *NewNameValue* and updates any uses of it in the model such as rules, events, reactions, variants, and doses to use the new name.

If the new name is already being used by another model component, the new name will be qualified to ensure that it is unique. For example if you change a species named *A* to *K*, and a parameter with the name *K* exists, the species will be qualified as *CompartmentName.K* to indicate that the reference is to the species. If you are referring to an object by its qualified name, for example *CompartmentName.A* and you change the species name, the reference will contain the qualified name in its updated form, for example, *CompartmentName.K*

When you want to change the name of a compartment, parameter, species, or reaction object, use this method instead of `set`.

---

**Note** The `set` method only changes the `Name` property of the object, except for species and compartments. The method updates the species or compartment object's `Name`

property and any reaction strings referring to the species or compartment to use the new name.

---

### Examples

- 1 Create a model object that contains a species A in a rule.

```
m = sbiomodel('cell');  
s = addspecies(m, 'A');  
r = addrule(m, 'A = 4');
```

- 2 Rename the species to Y

```
rename(s, 'Y');
```

- 3 See that the rule expression is now updated.

```
r  
  
SimBiology Rule Array  
  
Index:      RuleType:      Rule:  
1          initialAssignment  Y = 4
```

### See Also

set

**Introduced in R2008b**

## reorder (model, compartment, kinetic law)

Reorder component lists

### Syntax

```
modelObj = reorder(Obj,NewOrder)
```

### Input Arguments

Obj	Model, compartment, or kinetic law object.
NewOrder	Object vector in the new order. If Obj is a model object, NewOrder can be an array of compartment, event, parameter, reaction, rule, variant, or dose objects. If Obj is a compartment object, NewOrder must be an array of species objects. If Obj is a kinetic law object, NewOrder must be an array of parameter objects.
	<b>Warning</b> As of 2017b, reordering rules has no effect on simulation results because the rules are evaluated as a unified system of constraints. For details, see “Evaluation Order of Rules”.

### Description

`modelObj = reorder(Obj,NewOrder)` reorders the component vector `modelObj` to be in the order specified.

Use this method to reorder any of the component vectors, such as compartments, events, parameters, rules, species, doses, and variants. When reordered, the vector of components must contain the same objects as the original list of objects, though they can be in a different order.

## Examples

### Reorder Reactions in SimBiology Model

Import a model.

```
modelObj = sbmlimport('lotka');
```

Display reactions in the model.

```
modelObj.Reactions
```

```
SimBiology Reaction Array

Index:   Reaction:
1        x + y1 -> 2 y1 + x
2        y1 + y2 -> 2 y2
3        y2 -> z
```

Reverse the order of reactions in the model.

```
reorder(modelObj,modelObj.Reactions([3 2 1]));
```

Display the new order of reactions.

```
modelObj.Reactions
```

```
SimBiology Reaction Array

Index:   Reaction:
1        y2 -> z
2        y1 + y2 -> 2 y2
3        x + y1 -> 2 y1 + x
```

## See Also

Compartment object | KineticLaw object | Model object

**Introduced in R2007b**



# RepeatDose object

Define drug dosing protocol

## Description

A `RepeatDose` object defines a series of doses to the amount of a species during a simulation. The `TargetName` property of a dose object defines the species that receives the dose.

Each dose is the same amount, as defined by the `Amount` property, and given at equally spaced times, as defined by the `Interval` property. The `RepeatCount` property defines the number of injections in the series, excluding the initial injection. The `Rate` property defines how fast each dose is given.

To use a dose object in a simulation you must add the dose object to a model object and set the `Active` property of the dose object to true. Set the `Active` property to true if you always want the dose to be applied before simulating the model.

---

**Warning** The `Active` property of the `RepeatDose` object will be removed in a future release. Explicitly specify a dose or an array of doses as an input argument when you simulate a model using `sbiosimulate`.

---

When there are multiple active `RepeatDose` objects on a model and if there are duplicate specifications for a property value, the last occurrence for the property value in the array of dose, is used during simulation. You can find out which dose is applied last by looking at the indices of the dose objects stored on the model.

You can set these dose properties to model parameters: `Amount`, `Interval`, `Rate`, `RepeatCount`, `StartTime`, `LagParameterName` and `DurationParameterName`. You can set these properties, except `LagParameterName` and `DurationParameterName`, to either a numeric value or the name of a model-scoped parameter (as a character vector or string). Parameterizing dose properties provides more flexibility for different dosing applications, such as scaling the dose amount by body weight. For details, see “Parameterized and Adaptive Doses”.

You can also interactively change the dose properties using the SimBiology desktop.

## Constructor Summary

sbiodose                      Construct dose object

## Method Summary

Methods for RepeatDose objects

copyobj (any object)	Copy SimBiology object and its children
get (any object)	Get object properties
getTable(ScheduleDose,RepeatDose)	Return data from SimBiology dose object as table
set (any object)	Set object properties
setTable(ScheduleDose,RepeatDose)	Set dosing information from table to dose object

## Property Summary

Properties for RepeatDose objects

---

Active	Indicate object in use during simulation
Amount	Amount of dose
AmountUnits	Dose amount units
DurationParameterName	Parameter specifying length of time to administer a dose
EventMode	Determine how events that change dose parameters affect in-progress dosing
Interval	Time between doses
LagParameterName	Parameter specifying time lag for dose
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Rate	Rate of dose
RateUnits	Units for dose rate
RepeatCount	Dose repetitions
StartTime	Start time for initial dose time
Tag	Specify label for SimBiology object
TargetName	Species receiving dose
TimeUnits	Show time units for dosing and simulation
Type	Display SimBiology object type
UserData	Specify data to associate with object

## Examples

### Scale Dose Amount by Body Weight

Parameterize the Amount property of a dose to scale it by the body weight of a patient.

Create a simple model with linear elimination and an amount parameter.

```
model = sbiomodel('simple model');
compartment = addcompartment(model, 'Central', 1);
compartment.CapacityUnits = 'liter';
species = addspecies(model, 'drug');
```

```
species.InitialAmountUnits = 'milligram';

% Elimination rate
elimParam                  = addparameter(model, 'kel', 0.1);
elimParam.ValueUnits       = '1/hour';

% Elimination reaction
reaction                   = addreaction(model, 'drug -> null');
reaction.ReactionRate      = 'kel*drug';
amountParam                = addparameter(model, 'A', 50);
amountParam.ConstantValue = false;
amountParam.ValueUnits     = 'milligram'
```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	A	50	milligram

Create a dose with its Amount property set to the amount parameter 'A'.

```
dose                        = adddose(model, 'adaptive dose', 'repeat');
dose.Amount                 = 'A';
```

Set other dose properties.

```
dose.TargetName            = 'drug';
dose.StartTime              = 0;
dose.TimeUnits              = 'hour';
dose.Interval               = 24;
dose.RepeatCount            = 7;
```

Add a parameter to represent the body weight.

```
weightParam                = addparameter(model, 'weight', 80);
weightParam.ValueUnits     = 'kilogram';
```

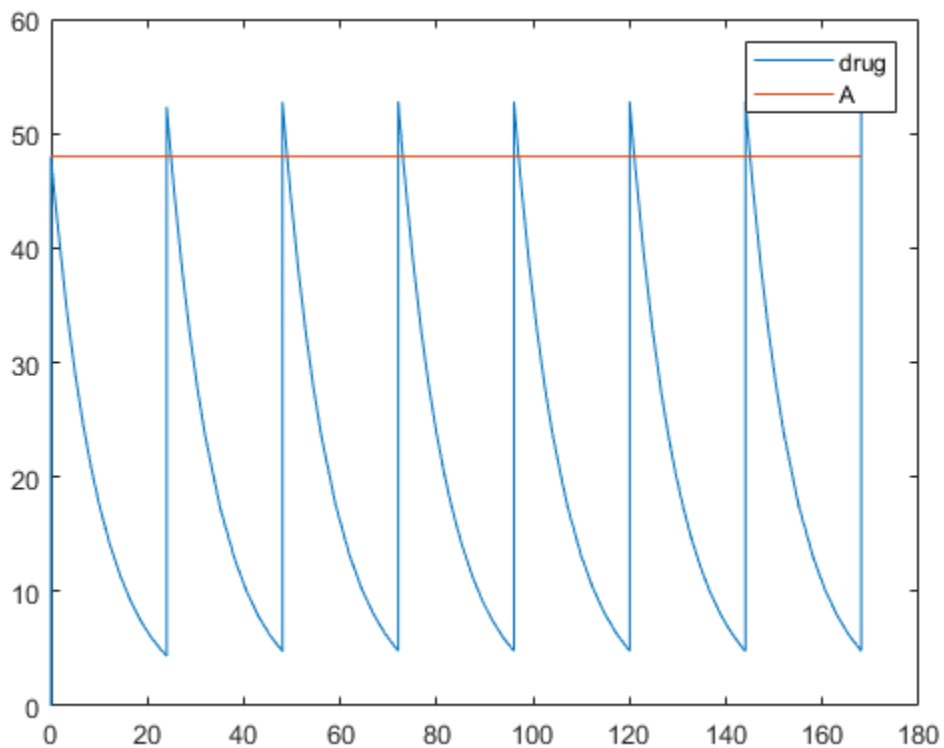
Scale the dose amount by the body weight using an initial assignment rule.

```
scaleParam                  = addparameter(model, 'doseAmountPerWeight', 0.6);
scaleParam.ValueUnits       = 'milligram/kilogram';
rule                         = addrule(model, 'A = weight*doseAmountPerWeight', 'initialAssignment');
```

Simulate the model for 7 days and plot the results.

```
configset                   = getconfigset(model);
configset.StopTime           = 7*24;
```

```
configset.TimeUnits = 'hour';  
[time, drugAndAmount] = sbiosimulate(model,dose);  
plot(time, drugAndAmount);  
legend('drug','A');
```



### Change Dose Behavior In Response to Changes in Model Parameters

Create a simple model with linear elimination, an amount parameter, and a rate parameter.

```

model = sbiomodel('simple model');
compartment = addcompartment(model, 'Central', 1);
compartment.CapacityUnits = 'liter';
species = addspecies(model, 'drug');
species.InitialAmountUnits = 'milligram';

```

*% Elimination rate*

```

elimParam = addparameter(model, 'kel', 0.1);
elimParam.ValueUnits = '1/hour';

```

*% Elimination reaction*

```

reaction = addreaction(model, 'drug -> null');
reaction.ReactionRate = 'kel*drug';

```

*% Add amount and rate parameters*

```

amountParam = addparameter(model, 'A', 50);
amountParam.ConstantValue = false;
amountParam.ValueUnits = 'milligram'

```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	A	50	milligram

```

rateParam = addparameter(model, 'R', 10);
rateParam.ValueUnits = 'milligram/hour'

```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	R	10	milligram/hour

Create a dose with its Amount and Rate properties set to the amount and rate parameters 'A' and 'R', respectively.

```

dose = adddose(model, 'adaptive dose', 'repeat');
dose.Amount = 'A';
dose.Rate = 'R';

```

Set other dose properties.

```

dose.TargetName = 'drug';
dose.StartTime = 0;
dose.TimeUnits = 'hour';
dose.Interval = 24;
dose.RepeatCount = 7;

```

Prepare the configuration set to simulate the model for 7 days.

```
configset          = getconfigset(model);  
configset.StopTime = 7*24;  
configset.TimeUnits = 'hour';
```

Add an event to reset the dose amount to 10 at time  $\geq 26$ .

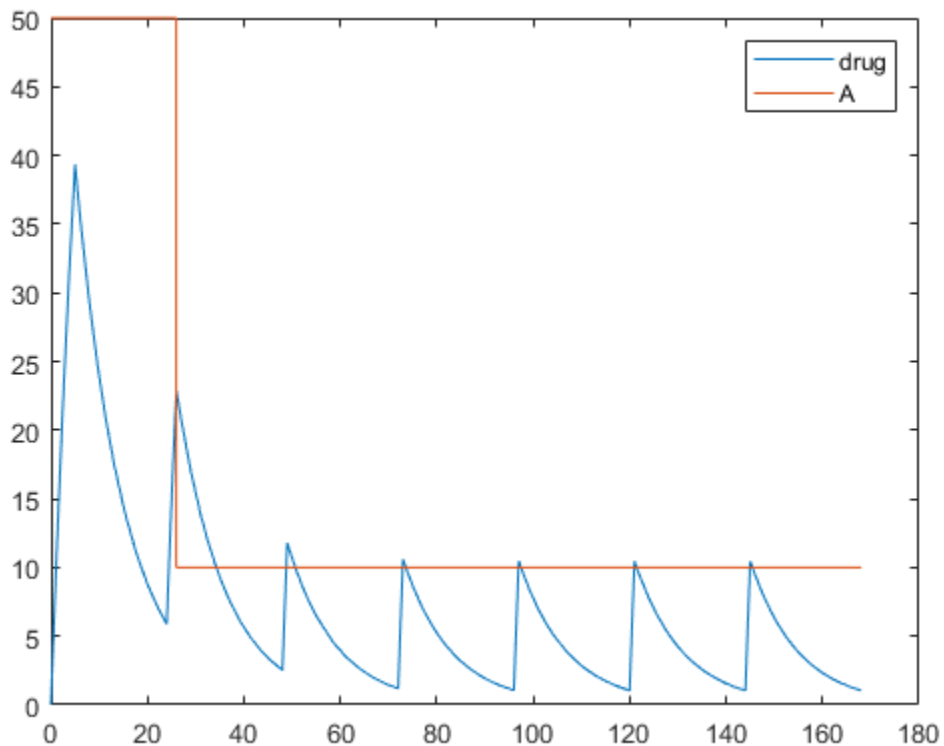
```
event = addevent(model, 'time  $\geq 26$ ', 'A = 10');
```

Set the EventMode property to 'stop'. This setting causes any ongoing dose event to stop at 26 hours.

```
dose.EventMode = 'stop';
```

Simulate the model. The second dose event stops at 26 hours, and the subsequent dose events continue with the new dose amount of 10.

```
[time, drugAndAmount] = sbiosimulate(model, dose);  
figure  
plot(time, drugAndAmount);  
legend('drug', 'A');
```



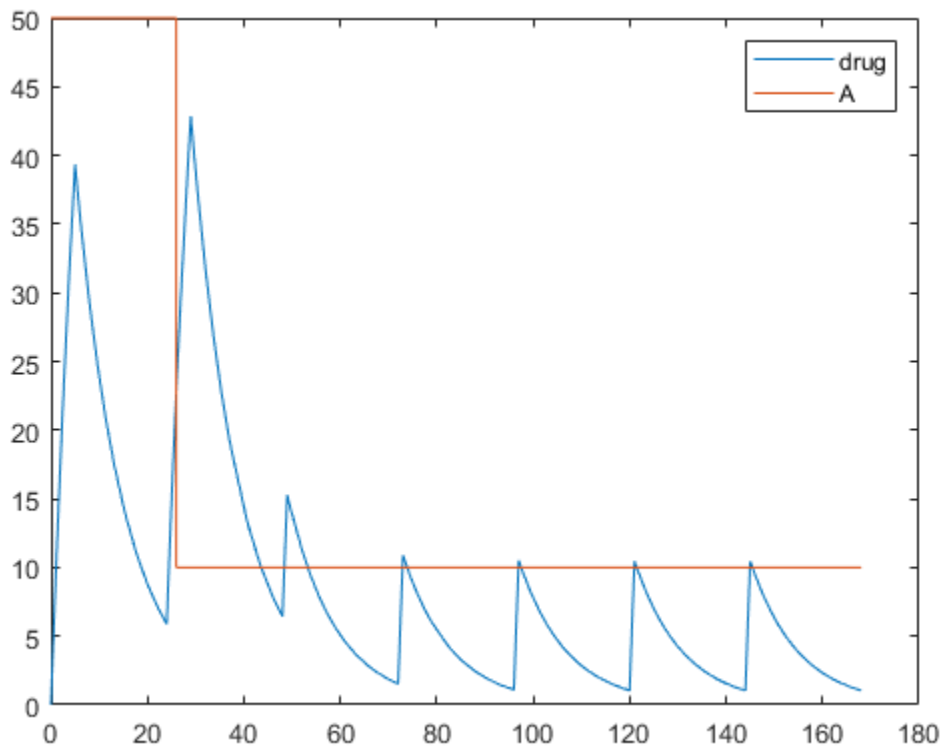
Alternatively, you can allow the ongoing dose event to finish before applying the new dose amount by setting `EventMode` to `'continue'`.

```
dose.EventMode = 'continue';
```

Simulate the model. In this case, the second dose event continues to 26 hours.

```
[time, drugAndAmount] = sbiosimulate(model,dose);  
figure  
plot(time, drugAndAmount);  
legend('drug', 'A');
```





## See Also

Model object | ScheduleDose object | sbiodose | sbiosimulate

## Topics

“Parameterized and Adaptive Doses”

**Introduced in R2010a**

## resample (SimData)

Resample SimData object array onto new time vector

### Syntax

```
newSimDataObj = resample(simDataObj)  
newSimDataObj = resample(simDataObj, timevector)  
newSimDataObj = resample(simDataObj, timevector, method)
```

### Arguments

<i>newSimDataObj</i>	Resampled SimData object array.
<i>simDataObj</i>	SimData object array that you want to resample.
<i>timevector</i>	Real numeric array of time points onto which you want to resample the data.

<i>method</i>	<p>Method to use during resampling. It can be one of the following:</p> <ul style="list-style-type: none"> <li>• 'interp1q' — Uses the interp1q function.</li> <li>• To use the interp1 function, specify one of the following methods: <ul style="list-style-type: none"> <li>• 'nearest'</li> <li>• 'linear'</li> <li>• 'spline'</li> <li>• 'pchip'</li> <li>• 'cubic'</li> <li>• 'v5cubic'</li> </ul> </li> <li>• 'zoh' — specifies zero-order hold.</li> </ul> <p>You can also specify the method as a string using double quotes, such as "pchip".</p> <hr/> <p><b>Warning:</b> Note that 'cubic' method will change in a future release. Use 'pchip' instead.</p>
---------------	---

## Description

*newSimDataObj* = *resample(simDataObj)* resamples the simulation data contained in every element of the SimData object array *simDataObj* onto a common time vector, producing a new SimData array *newSimDataObj*. By default, the common time vector is taken from the element of *simDataObj* with the earliest stopping time.

*newSimDataObj* = *resample(simDataObj, timevector)* resamples the SimData array *simDataObj* onto the time vector *timevector*. *timevector* must either be a real numeric array or the empty array []. If you use an empty array, *resample* uses the default time vector as described above.

*newSimDataObj* = *resample(simDataObj, timevector, method)* uses the interpolation method specified in *method*.

If the specified *timevector* includes time points outside the time interval encompassed by one or more SimData objects in *simDataObj*, the resampling will involve extrapolation

and you will see a warning. See the help for the MATLAB function corresponding to the interpolation method in use for information on how the function performs the extrapolation.

## Examples

### Simulating and Resampling Data

- 1 The project file, `radiodecay.sbproj` contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.

```
sbioloadproject('radiodecay');  
simDataObj = sbiosimulate(m1);
```

- 2 Resample the data.

```
newSimDataObj = resample(simDataObj, [1:5], 'linear');
```

### Resampling Data for Ensemble Runs

- 1 The project file, `radiodecay.sbproj`, contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.

```
sbioloadproject('radiodecay');
```

- 2 Change the solver to use during the simulation and perform an ensemble run.

```
csObj = getconfigset(m1);  
set(csObj, 'SolverType', 'ssa');  
simDataObj = sbioenssemblerun(m1, 10);
```

- 3 Interpolate the time steps.

```
newSimDataObj = resample(simDataObj, [1:10], 'linear');
```

- 4 View the time steps in the `SimData` object arrays.

```
newSimDataObj(1).Time  
simDataObj(1).Time
```

## See Also

`sbioensamplerun`, `sbioensemblestats`, `sbiosimulate`, `SimData` object

MATLAB functions `interp1`, `interp1q`

**Introduced in R2007b**

## reset (root)

Delete all model objects from root object

### Syntax

```
reset(sbioroot)
```

### Description

`reset(sbioroot)` deletes all SimBiology model objects contained by the root object. This call is equivalent to `sbioreset` on page 1-250.

The root object contains a list of model objects, available units, unit prefixes, and kinetic laws.

To add a kinetic law to the user-defined library, use the `sbioaddtolibrary` function. To add a unit to the user-defined library, use `sbiounit` followed by `sbioaddtolibrary`. To add a unit prefix to the user-defined library, use `sbiounitprefix` followed by `sbioaddtolibrary`.

### Examples

- 1 Query `sbioroot`, which has two model objects.

```
sbioroot
```

```
SimBiology Root Contains:
```

```
Models:                2
Builtin Abstract Kinetic Laws:  3
User Abstract Kinetic Laws:    1
Builtin Units:           54
User Units:              0
Builtin Unit Prefixes:   13
User Unit Prefixes:     0
```

## 2 Call reset.

sbioroot

SimBiology Root Contains:

Models:	0
Builtin Abstract Kinetic Laws:	3
User Abstract Kinetic Laws:	1
Builtin Units:	54
User Units:	0
Builtin Unit Prefixes:	13
User Unit Prefixes:	0

## See Also

sbioaddtolibrary, sbioreset, sbioroot, sbiounit, sbiounitprefix

**Introduced in R2006a**

## rmcontent (variant)

Remove contents from variant object

### Syntax

```
rmcontent(variantObj, contents)
```

```
rmcontent(variantObj, idx)
```

### Arguments

<i>variantObj</i>	Specify the variant object from which you want to remove data. The Content property is modified to remove the new data.
-------------------	---



<i>contents</i>	Specify the data you want to remove from a variant object. Contents can either be a cell array or an array of cell arrays. A valid cell array should have the form { 'Type', 'Name', 'PropertyName', PropertyValue }, where PropertyValue is the new value to be applied for the PropertyName. Valid Type, Name, and PropertyName values are as follows.		
	<b>'Type'</b>	<b>'Name'</b>	<b>'PropertyName'</b>
	'species'	Name of the species. If there are multiple species in the model with the same name, specify the species as [compartmentName.speciesName], where compartmentName is the name of the compartment containing the species.	'InitialAmount'
	'parameter'	If the parameter scope is a model, specify the parameter name. If the parameter scope is a kinetic law, specify [reactionName.parameterName].	'Value'
	'compartment'	Name of the compartment.	'Capacity'
<i>idx</i>	Specify the ContentIndex or indices of the data to be removed. To display the ContentIndex, enter the object name and press <b>Enter</b> .		

## Description

`rmcontent(variantObj, contents)` removes the data stored in the variable *contents* from the variant object (*variantObj*).

`rmcontent(variantObj, idx)` removes the data specified by the indices *idx* (also called ContentIndex) from the Content property of the variant object.

## Examples

- 1 Create a model containing three species in one compartment.

```
modelObj = sbiomodel('mymodel');
compObj = addcompartment(modelObj, 'comp1');
A = addspecies(compObj, 'A');
B = addspecies(compObj, 'B');
C = addspecies(compObj, 'C');
```

- 2 Add a variant object that varies the species' InitialAmount property.

```
variantObj = addvariant(modelObj, 'v1');
addcontent(variantObj, {'species','A', 'InitialAmount', 5}, ...
{'species', 'B', 'InitialAmount', 10}, ...
{'species', 'C', 'InitialAmount', 15});% Display the variant
variantObj
```

```
SimBiology Variant - v1 (inactive)
```

ContentIndex:	Type:	Name:	Property:	Value:
1	species	A	InitialAmount	5
2	species	B	InitialAmount	10
3	species	C	InitialAmount	15

- 3 Use the ContentIndex number to remove a species from the Content property of the variant object.

```
rmcontent(variantObj, 2);
variantObj
```

```
SimBiology Variant - v1 (inactive)
```

ContentIndex:	Type:	Name:	Property:	Value:
1	species	A	InitialAmount	5
2	species	C	InitialAmount	15

- 4 (Alternatively) Remove a species from the contents of the variant object using detailed reference to the species.

```
rmcontent(variantObj, {'species','A', 'InitialAmount', 5});
% Display variant object
variantObj
SimBiology Variant - v1 (inactive)
```

ContentIndex:	Type:	Name:	Property:	Value:
1	species	C	InitialAmount	15

## **See Also**

addvariant, rmcontent, sbiovariant

**Introduced in R2007b**

## rmproduct (reaction)

Remove species object from reaction object products

### Syntax

```
rmproduct(reactionObj, SpeciesName)  
rmproduct(reactionObj, speciesObj)
```

### Arguments

<i>reactionObj</i>	Reaction object.
<i>SpeciesName</i>	Name for a model object. Enter a species name or a cell array of species names.
<i>speciesObj</i>	Species object. Enter a species object or an array of species objects.

### Description

`rmproduct(reactionObj, SpeciesName)`, in a reaction object (*reactionObj*), removes a species object with a specified name (*SpeciesName*) from the property *Products*, removes the species name from the property *Reaction*, and updates the property *Stoichiometry* to exclude the species coefficient.

`rmproduct(reactionObj, speciesObj)` removes a species object as described above using a MATLAB variable for a species object.

The species object is not removed from the parent model property *Species*. If the species object is no longer used by any reaction, you can use the function `delete` to remove it from the parent object.

If one of the species specified does not exist as a product, a warning is returned.

## Examples

### Example 1

This example shows how to remove a product that was previously added to a reaction. You can remove the species object using the species name.

```
modelObj = sbiomodel('cell');
reactionObj = addreaction(modelObj, 'Phosphocreatine + ADP -> creatine + ATP + Pi');
rmproduct(reactionObj, 'Pi')
```

SimBiology Reaction Array

```
Index: Reaction:
1      Phosphocreatine + ADP -> creatine + ATP
```

### Example 2

Remove a species object using a model index to a species object.

```
modelObj = sbiomodel('cell');
reactionObj = addreaction(modelObj, 'A -> B + C');
reactionObj.Reaction
ans =
    A -> B + C

rmproduct(reactionObj, modelObj.Species(2));
reactionObj.Reaction
ans =
    A -> C
```

## See Also

rmreactant

**Introduced in R2006a**

## rmreactant (reaction)

Remove species object from reaction object reactants

### Syntax

```
rmreactant(reactionObj, SpeciesName)  
rmreactant(reactionObj, speciesObj)
```

### Arguments

<i>reactionObj</i>	Reaction object.
<i>SpeciesName</i>	Name for a species object. Enter a species name or a cell array of species names.
<i>speciesObj</i>	Species object. Enter a species object or an array of species objects.

### Description

`rmreactant(reactionObj, SpeciesName)`, in a reaction object (*reactionObj*), removes a species object with a specified name (*SpeciesName*) from the property `Reactants`, removes the species name from the property `Reaction`, and updates the property `Stoichiometry` to exclude the species coefficient.

`rmreactant(reactionObj, speciesObj)` removes a species object as described above using a MATLAB variable for a species object, or a model index for a species object.

The species object is not removed from the parent model property `Species`. If the species object is no longer used by any reaction, you can use the method `delete` to remove it from the parent object.

If one of the species specified does not exist as a reactant, a warning is returned.

## Examples

### Example 1

This example shows how to remove a reactant that was added to a reaction by mistake. You can remove the species object using the species name.

```
modelObj = sbiomodel('cell');
reactionObj = addreaction(modelObj, 'Phosphocreatine + ADP + Pi -> creatine + ATP');
rmreactant(reactionObj, 'Pi')
```

SimBiology Reaction Array

Index:	Reaction:
1	Phosphocreatine + ADP -> creatine + ATP

### Example 2

Remove a species object using a model index to a species object.

```
modelObj = sbiomodel('cell');
reactionObj = addreaction(modelObj, 'A -> B + C');
```

```
reactionObj.Reaction
```

```
ans =
    A + B -> C
```

```
rmreactant(reactionObj, modelObj.Species(1));
reactionObj.Reaction
```

```
ans =
    A -> C
```

## See Also

delete, rmproduct

**Introduced in R2006a**

## Root object

Hold models, unit libraries, and abstract kinetic law libraries

### Description

The SimBiology root object contains a list of the SimBiology model objects and SimBiology libraries. The components that the libraries contain are: all available units, unit prefixes, and available abstract kinetic law objects. There are two types of libraries: one contains components that are built in (`BuiltinLibrary`), and the other contains components that are user defined (`UserdefinedLibrary`).

You can retrieve SimBiology model objects from the SimBiology root object. A SimBiology model object has its `Parent` property set to the SimBiology root object.

See “Property Summary” on page 2-429 for links to root object property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can interactively change object properties in the SimBiology desktop.

### Constructor Summary

<code>sbioroot</code>	Return SimBiology root object
-----------------------	-------------------------------

### Method Summary

<code>copyobj</code> (any object)	Copy SimBiology object and its children
<code>get</code> (any object)	Get object properties
<code>reset</code> (root)	Delete all model objects from root object
<code>set</code> (any object)	Set object properties



## Property Summary

BuiltInLibrary	Library of built-in components
Models	Contain all model objects
Type	Display SimBiology object type
UserDefinedLibrary	Library of user-defined components

## See Also

AbstractKineticLaw object, Configset object, KineticLaw object, Model object, Parameter object, Reaction object, Rule object, Species object

**Introduced in R2006b**

## Rule object

Hold rule for species and parameters

### Description

The SimBiology rule object represents a *rule*, which is a mathematical expression that modifies a species amount or a parameter value. For a description of the types of SimBiology rules, see `RuleType`.

See “Property Summary” on page 2-431 for links to rule property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the graphical user interface.

### Constructor Summary

`addrule (model)`      Create rule object and add to model object

### Method Summary

<code>copyobj (any object)</code>	Copy SimBiology object and its children
<code>delete (any object)</code>	Delete SimBiology object
<code>display (any object)</code>	Display summary of SimBiology object
<code>get (any object)</code>	Get object properties
<code>set (any object)</code>	Set object properties

## Property Summary

Active	Indicate object in use during simulation
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Rule	Specify species and parameter interactions
RuleType	Specify type of rule for rule object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

“Definitions and Evaluations of Rules”, `AbstractKineticLaw` object, `Configset` object, `KineticLaw` object, `Model` object, `Parameter` object, `Reaction` object, `Root` object, `Species` object

**Introduced in R2006b**

## summary(LeastSquaresResults, OptimResults, NLINResults)

Plot a summary figure that contains estimated values and estimation statistics

### Syntax

```
summary(resultsObj)
```

### Description

`summary(resultsObj)` plots a summary figure that contains estimated values and estimation statistics.

### Input Arguments

#### **resultsObj** — Estimation results

`OptimResults` object | `NLINResults` object | vector of results objects

Estimation results, specified as an `OptimResults` object or `NLINResults` object, or vector of results objects which contains estimation results from running `sbiofit`.

### See Also

`NLINResults` object | `OptimResults` object | `sbiofit`

**Introduced in R2014a**

## ScheduleDose object

Define drug dosing protocol

### Description

A `ScheduleDose` object defines a series of doses to the amount of a species during a simulation. The `TargetName` property defines the species that receives the dose.

Each dose can have a different amount, as defined by an amount array in the `Amount` property. Each dose can be given at specified times, as defined by a time array in the `Time` property. A rate array in the `Rate` property defines how fast each dose is given. At each time point in the time array, a dose is given with the corresponding amount and rate.

To use a dose object in a simulation you must add the dose object to a model object and set the `Active` property of the dose object to true. Set the `Active` property to true if you always want the dose to be applied before simulating the model.

---

**Warning** The `Active` property of the `ScheduleDose` object will be removed in a future release. Explicitly specify a dose or an array of doses as an input argument when you simulate a model using `sbiosimulate`.

---

When there are multiple active `ScheduleDose` objects on a model, and there are duplicate specifications for a property value, the simulation uses the last occurrence for the property value in the array of doses. You can find out which dose you applied last by looking at the indices of the dose objects stored on the model.

See the “Property Summary” on page 2-505 for links to species property reference pages. Properties define the characteristics of an object. Use the `get` command to list object properties and the `set` command to change their values at the command line.

You can also interactively change the dose properties using the SimBiology desktop.

---

**Tip** You can create a combination of bolus and infusion doses by setting the `rate` property of a `ScheduleDose` object to a vector containing zeros and non-zeros.

---

## Constructor Summary

sbiodose                      Construct dose object

## Method Summary

Methods for variant objects

copyobj (any object)	Copy SimBiology object and its children
get (any object)	Get object properties
getTable(ScheduleDose,RepeatDose)	Return data from SimBiology dose object as table
set (any object)	Set object properties
setTable(ScheduleDose,RepeatDose)	Set dosing information from table to dose object

## Property Summary

Properties for variant objects

Active	Indicate object in use during simulation
Amount	Amount of dose
AmountUnits	Dose amount units
DurationParameterName	Parameter specifying length of time to administer a dose
EventMode	Determine how events that change dose parameters affect in-progress dosing
LagParameterName	Parameter specifying time lag for dose
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Rate	Rate of dose
RateUnits	Units for dose rate
Tag	Specify label for SimBiology object
TargetName	Species receiving dose
Time	Simulation time steps or schedule dose times
TimeUnits	Show time units for dosing and simulation
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

Model object, RepeatDose object, sbiodose, sbiosimulate

**Introduced in R2010a**

## select (SimData)

Select data from SimData object

### Syntax

```
[t,x, names] = select(simDataObj, Query)
[Out] = select(simDataObj, Query, 'Format', 'FormatValue')
```

### Arguments

#### Output Arguments

<i>t</i>	An n-by-1 vector of time points.
<i>x</i>	An n-by-m data array. <i>t</i> and <i>names</i> label the rows and columns of <i>x</i> respectively.
<i>names</i>	An m-by-1 cell array of names.
<i>Out</i>	Data returned in the format specified in ' <i>FormatValue</i> ', shown in "Input Arguments" on page 2-436. Depending on the specified ' <i>FormatValue</i> ', <i>Out</i> contains one of the following: <ul style="list-style-type: none"><li>• Structure array</li><li>• SimData object</li><li>• Time series object</li><li>• Combined time series object from an array of SimData objects</li></ul>

#### Input Arguments

<i>simDataObj</i>	SimData object array. Enter a variable name for a SimData object.
-------------------	---



<i>Query</i>	A cell array of arguments consisting of some combination of property name/property value pairs and/or 'Where' clauses. For a more complete description of the query syntax, including 'Where' clauses and their supported condition types, see <code>sbiiselect</code> . You can use any of the metadata fields available in the cells of the <code>DataInfo</code> property of a <code>SimData</code> object in a query. These include 'Type', 'Name', 'Units', 'Compartment' (species only), or 'Reaction' (parameter only).
<i>FormatValue</i>	Choose a format from the following table.

<b>FormatValue</b>	<b>Description</b>
'num'	Specifies the format that lets you return data in numeric arrays. This is the default when <code>select</code> is called with multiple output arguments.
'nummetadata'	Specifies the format that lets you return a cell array of metadata structures in <i>metadata</i> instead of names. The elements of <i>metadata</i> label the columns of <i>x</i> .
'numqualnames'	Specifies the format that lets you return qualified names in <i>names</i> to resolve ambiguities.
'struct'	Specifies the format that lets you return a structure array holding both data and metadata. This is the default when you use a single output argument.
'simdata'	Specifies the format that lets you return data in a new <code>SimData</code> object. This is the default format when <code>select</code> is called with zero or one output argument.
'ts'	Specifies the format that lets you return data in time series objects, creating an individual time series for each state or column and <code>SimData</code> object in <code>simDataObj</code> .
'tslumped'	Specifies the format that lets you return data in time series objects, combining data from each <code>SimData</code> object into a single time series.

## Description

`[t,x,names] = select(simDataObj, Query)` returns simulation time and state data from the `SimData` object (`simDataObj`) that matches the query argument *Query*.

In a `SimData` object `simDataObj`, the columns of the data matrix `simDataObj.Data` are labeled by the cell array of metadata structures given by `simDataObj.DataInfo`. The `select` method enables you to pick out columns of the data matrix based on their metadata labels. For example, to extract data for all parameters logged in a `SimData` object `simDataObj`, use the syntax `[t, x, names] = select(simDataObj, {'Type', 'parameter'})`.

`[Out] = select(simDataObj, Query, 'Format', 'FormatValue')` returns the data in the specified format. Valid formats are listed in “Input Arguments” on page 2-436.

## Examples

This example shows how to extract data of interest from your simulation data with the `select` method.

- 1 The project file `radiodecay.sbproj` contains a model stored in a variable called `m1`. Load `m1` into the MATLAB workspace.

```
sbioloadproject gprotein_norules m1
```

- 2 Change the solver to use during the simulation and perform an ensemble run.

```
csObj = getconfigset(m1);  
set(csObj, 'SolverType', 'ssa');  
simDataObj = sbioensemblerun(m1, 10);
```

- 3 Select all species data logged in the `SimData` array `sdataarray`.

```
[t x n] = select(simDataObj, {'Type', 'species'});
```

- 4 Select data for the parameters with name 'Kd' and return the results in a new `SimData` object array.

```
newsd = select(simDataObj, {'Type', 'parameter', 'name', 'Kd'});
```

- 5 This selects all data from `simDataObj` with a name that matches the pattern 'G' and returns time series objects.

```
ts = select(simDataObj, {'Where', 'Name', 'regexp', 'G'}, ...  
            'Format', 'ts');
```

## See Also

getdata, sbioselect, sbiosimulate, selectbyname, Simdata object

**Introduced in R2007b**

## selectbyname (SimData)

Select data by name from SimData object array

### Syntax

```
[t,x,n] = selectbyname(simDataObj, 'NameValue')  
Out = selectbyname(simDataObj, NameValue, 'Format', Format)
```

### Arguments

#### Output Arguments

<i>t</i>	An <i>n</i> -by-1 vector of time points.
<i>x</i>	An <i>n</i> -by- <i>m</i> data array. <i>t</i> and <i>names</i> label the rows and columns of <i>x</i> respectively.
<i>n</i>	An <i>m</i> -by-1 cell array of names.
<i>Out</i>	Data returned in the format as specified in ' <i>FormatValue</i> ', shown in "Input Arguments" on page 2-440. Depending on the specified ' <i>FormatValue</i> ', <i>Out</i> contains one of the following: <ul style="list-style-type: none"><li>• Structure array</li><li>• SimData object</li><li>• Time series object</li><li>• Combined time series object from an array of SimData objects</li></ul>

#### Input Arguments

<i>simDataObj</i>	SimData object array. Enter a variable name for a SimData object.
<i>NameValue</i>	Names of the states for which you want to select data from <i>simDataObj</i> . It must be a character vector, string, string vector, or cell array of character vectors.

<i>Query</i>	A cell array of arguments consisting of some combination of property name/property value pairs and/or 'Where' clauses. For a more complete description of the query syntax, including 'Where' clauses and their supported condition types, see <code>sbiselect</code> . You can use any of the metadata fields available in the cells of the <code>DataInfo</code> property of a <code>SimData</code> object. These include 'Type', 'Name', 'Units', 'Compartment' (species only), or 'Reaction' (parameter only).
<i>FormatValue</i>	Choose a format from the following table.

<b>FormatValue</b>	<b>Description</b>
'num'	Specifies the format that lets you return data in numeric arrays. This is the default when <code>selectbyname</code> is called with multiple output arguments.
'nummetadata'	Specifies the format that lets you return a cell array of metadata structures in <i>metadata</i> instead of names. The elements of <i>metadata</i> label the columns of <i>x</i> .
'numqualnames'	Specifies the format that lets you return qualified names in <i>names</i> to resolve ambiguities.
'struct'	Specifies the format that lets you return a structure array holding both data and metadata. This is the default when you use a single output argument.
'simdata'	Specifies the format that lets you return data in a new <code>SimData</code> object. This is the default format when <code>selectbyname</code> is called with zero or one output argument.
'ts'	Specifies the format that lets you return data in time series objects, creating an individual time series for each state or column and <code>SimData</code> object in <code>simDataObj</code> .
'tslumped'	Specifies the format that lets you return data in time series objects, combining data from each <code>SimData</code> object into a single time series.

## Description

The `selectbyname` method allows you to select data from a `SimData` object array by name. `[t,x,n] = selectbyname(simDataObj, 'NameValue')` returns time and state data from the `SimData` object `simDataObj` for states with names `'NameValue'`.

In a `SimData` object `simDataObj`, the names labeling the columns of the data matrix `simDataObj.Data` are given by `simDataObj.DataNames`. A name specified in `'NameValue'` can match more than one data column, for example, when `simDataObj` contains data for a species and parameter both named `'k'`. To resolve ambiguities, use qualified names in `'NameValue'`, such as `'CompartmentName.SpeciesName'` or `'ReactionName.ParameterName'`. `selectbyname` returns qualified names in the output argument `names` when there are ambiguities.

`Out = selectbyname(simDataObj, NameValue, 'Format', Format)` returns the data in the specified format. Valid formats are listed in “Input Arguments” on page 2-440.

## Examples

### Select Specific Species Data from Simulation Results

This example uses the Lotka-Volterra (predator-prey) model described by Gillespie [1].

Load the sample project containing the Lotka-Volterra model `m1`.

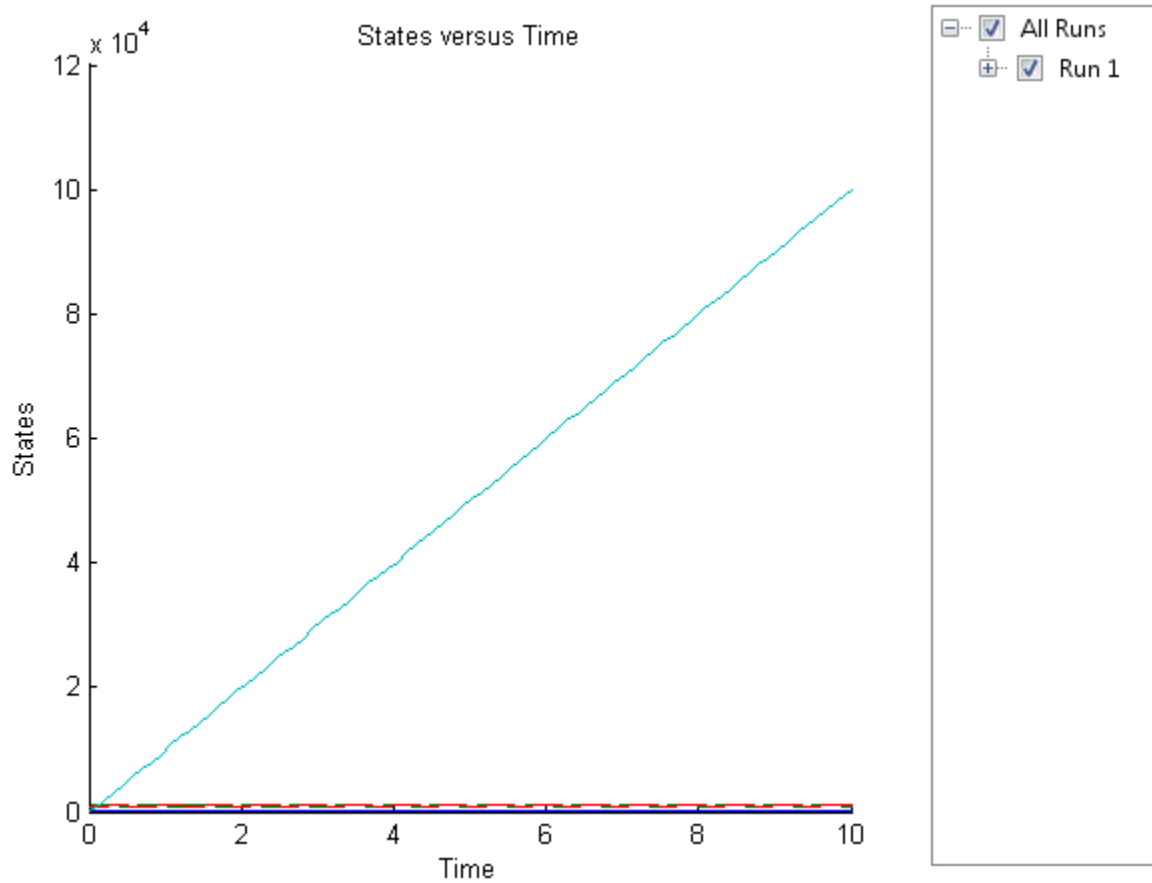
```
sbioloadproject lotka;
```

Simulate the model `m1`.

```
sd = sbiosimulate(m1);
```

Plot the simulation results which show the states of all species in the model.

```
sbioplot(sd);
```

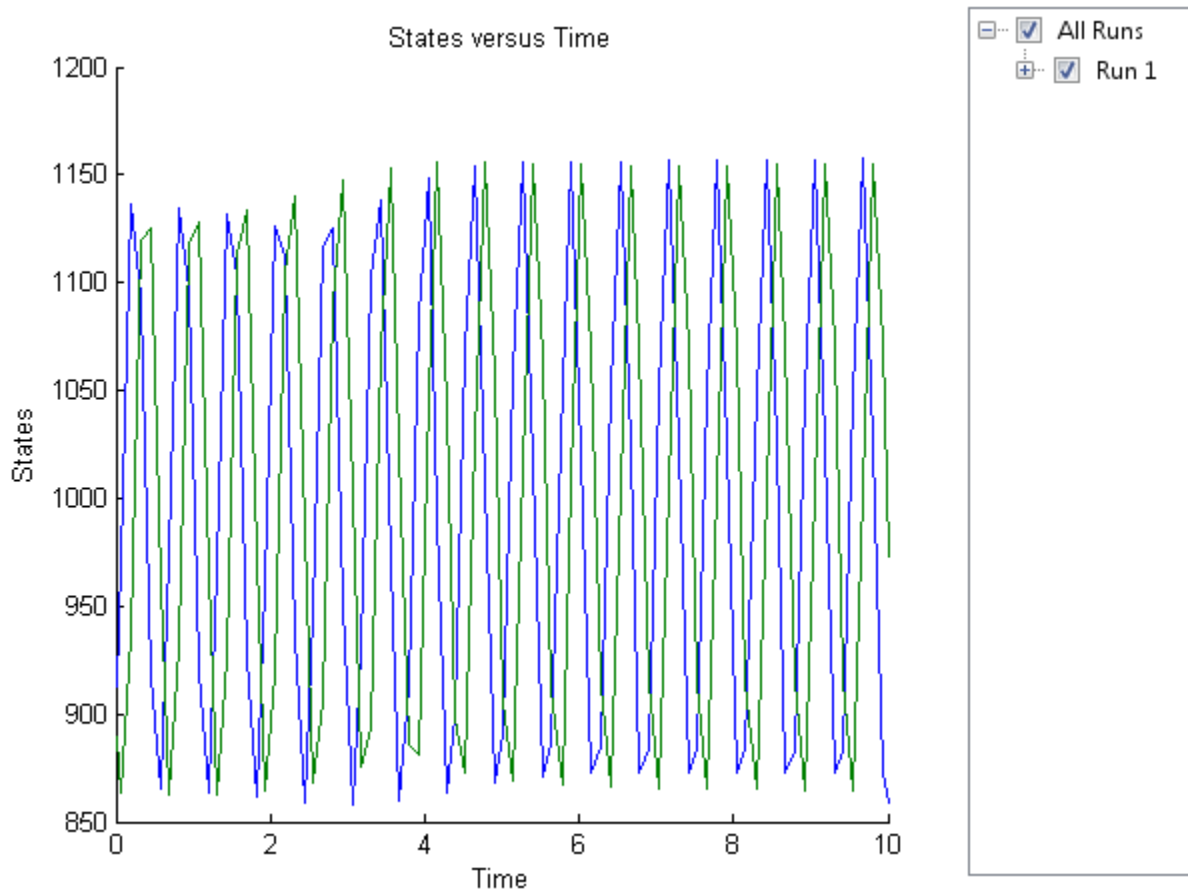


Select the simulation data of y1 and y2 species only.

```
sd2 = selectbyname(sd, {'y1', 'y2'});
```

Plot the selected data.

```
sbioplot(sd2);
```



## See Also

[1] Gillespie D.T. "Exact Stochastic Simulation of Coupled Chemical Reactions," (1977) The Journal of Physical Chemistry, 81(25), 2340-2361.

`getdata`, `sbioselect`, `sbiosimulate`

**Introduced in R2007b**



## set (any object)

Set object properties

### Syntax

```
set(Obj, 'PropertyName', PropertyValue)
set(Obj, 'PropertyName1', PropertyValue1, 'PropertyName2',
PropertyValue2...)
```

### Arguments

<i>Obj</i>	Abstract kinetic law, compartment, configuration set, event, kinetic law, model, parameter, PKCompartment, PKData, PKModelDesign PKModelMap, reaction, rule, SimData, species, or variant object.
' <i>PropertyName</i> '	Name of the property to set.
<i>PropertyValue</i>	Specify the value to set. Property values depend on the property being set. See the reference page for an object property for values that can be specified.

### Description

`set(Obj, 'PropertyName', PropertyValue)` sets the property '*PropertyName*' of the object *Obj*, to *PropertyValue*.

`set(Obj, 'PropertyName1', PropertyValue1, 'PropertyName2', PropertyValue2...)` sets the properties '*PropertyName1*' and '*PropertyName2*' to *PropertyValue1* and *PropertyValue2* respectively, and so on in sequence. You can specify multiple *PropertyName*, *PropertyValue* pairs.

When you want to change the name of a compartment, parameter, or species object, use the `rename` method instead of `set`. The `rename` method allows you to change the name and update the expressions in which these components are used.

### Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add parameter object.

```
parameterObj = addparameter (modelObj, 'kf');
```

- 3 Set the ConstantValue property of the parameter object to false and verify.

MATLAB returns 1 for true and 0 for false.

```
set (parameterObj, 'ConstantValue', false);  
get(parameterObj, 'ConstantValue')
```

MATLAB returns

```
ans =
```

```
0
```

### See Also

`get`, `rename`, `setactiveconfigset`

**Introduced in R2008b**

## setactiveconfigset (model)

Set active configuration set for model object

### Syntax

```
configsetObj = setactiveconfigset(modelObj, 'NameValue')  
configsetObj2 = setactiveconfigset(modelObj, configsetObj1)
```

### Description

*configsetObj* = setactiveconfigset(*modelObj*, 'NameValue') sets the configuration set *NameValue* to be the active configuration set for the model object *modelObj* and returns to *configsetObj*.

*configsetObj2* = setactiveconfigset(*modelObj*, *configsetObj1*) sets the configset *configsetObj1* to be the active configset for *modelObj* and returns to *configsetObj2*. Any change in one of these two configset objects *configsetObj1* and *configsetObj2* is reflected in the other. To copy over a configset object from one model object to another, use the copyobj method.

The active configuration set contains the settings that are be used during a simulation. A default configuration set is attached to any new model.

### Examples

- 1 Create a model object by importing the `oscillator.xml` file, and add a Configset object to the model.

```
modelObj = sbmlimport('oscillator');  
configsetObj = addconfigset(modelObj, 'myset');
```

- 2 Configure the simulation stop criteria by setting the `StopTime`, `MaximumNumberOfLogs`, and `MaximumWallClock` properties of the Configset object. Set the stop criteria to a simulation time of 3000 seconds, 50 logs, or a wall clock time of 10 seconds, whichever comes first.

```
set(configsetObj, 'StopTime', 3000, 'MaximumNumberOfLogs', 50,...  
    'MaximumWallClock', 10)  
get(configsetObj)
```

```
        Active: 0  
        CompileOptions: [1x1 SimBiology.CompileOptions]  
            Name: 'myset'  
            Notes: ''  
        RuntimeOptions: [1x1 SimBiology.RuntimeOptions]  
SensitivityAnalysisOptions: [1x1 SimBiology.SensitivityAnalysisOptions]  
        SolverOptions: [1x1 SimBiology.ODESolverOptions]  
            SolverType: 'ode15s'  
            StopTime: 3000  
MaximumNumberOfLogs: 50  
MaximumWallClock: 10  
        TimeUnits: 'second'  
        Type: 'configset'
```

- 3 Set the new Configset object to be active, simulate the model using the new Configset object, and plot the result.

```
    setactiveconfigset(modelObj, configsetObj);  
    [t,x] = sbiosimulate(modelObj);  
    plot (t,x)
```

## See Also

Model object, addconfigset, getconfigset, removeconfigset

**Introduced in R2006a**

## setparameter (kineticlaw)

Specify specific parameters in kinetic law object

### Syntax

```
setparameter(kineticlawObj, 'ParameterVariablesValue',
'ParameterVariableNamesValue')
```

### Arguments

<i>ParameterVariableValue</i>	Specify the value of the parameter variable in the kinetic law object.
<i>ParameterVariableNamesValue</i>	Specify the parameter name with which to configure the parameter variable in the kinetic law object. Determines parameters in the ReactionRate equation.

### Description

Configure ParameterVariableNames in the kinetic law object.

setparameter(*kineticlawObj*, 'ParameterVariablesValue', 'ParameterVariableNamesValue') configures the ParameterVariableNames property of the kinetic law object (*kineticlawObj*). ParameterVariableValue corresponds to one of the character vectors in *kineticlawObj* ParameterVariables property. The corresponding element in the *kineticlawObj*ParameterVariableNames property is configured to ParameterVariableNamesValue. For example, if ParameterVariables is {'Vm', 'Km'} and ParameterVariablesValue is specified as Vm, then the first element of the ParameterVariableNames cell array is configured to ParameterVariableNamesValue.

## Examples

Create a model, add a reaction, and then assign the `ParameterVariableNames` for the reaction rate equation.

- 1 Create the model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

`reactionObj` `KineticLaw` property is configured to `kineticlawObj`.

- 3 The 'Henri-Michaelis-Menten' kinetic law has two parameter variables (`Vm` and `Km`) that should be set. To set these variables:

```
setparameter(kineticlawObj, 'Vm', 'Va');  
setparameter(kineticlawObj, 'Km', 'Ka');
```

- 4 Verify that the parameter variables are correct.

```
get(kineticlawObj, 'ParameterVariableNames')
```

MATLAB returns:

```
ans =
```

```
    'Va'    'Ka'
```

## See Also

`addparameter`, `getspecies`, `setspecies`

**Introduced in R2006a**

## setspecies (kineticlaw)

Specify species in kinetic law object

### Syntax

```
setspecies(kineticlawObj, 'SpeciesVariablesValue',  
'SpeciesVariableNamesValue')
```

### Arguments

<i>SpeciesVariablesValue</i>	Specify the species variable in the kinetic law object.
<i>SpeciesVariableNamesValue</i>	Specify the species name with which to configure the species variable in the kinetic law object. Determines the species in the ReactionRate equation.

### Description

setspecies configures the kinetic law object SpeciesVariableNames property.

setspecies(kineticlawObj, 'SpeciesVariablesValue', 'SpeciesVariableNamesValue') configures the SpeciesVariableNames property of the kinetic law object, kineticlawObj. SpeciesVariablesValue corresponds to one of the character vectors in the SpeciesVariables property of kineticlawObj. The corresponding element in kineticlawObj SpeciesVariableNames property is configured to SpeciesVariableNamesValue.

For example, if SpeciesVariables are {'S', 'S1'} and SpeciesVariablesValue is specified as S1, the first element of the SpeciesVariableNames cell array is configured to SpeciesVariableNamesValue.

## Examples

Create a model, add a reaction, and assign the `SpeciesVariableNames` for the reaction rate equation.

- 1 Create the model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

reactionObj.KineticLaw property is configured to kineticlawObj.

- 3 The 'Henri-Michaelis-Menten' kinetic law has one species variable (S) that should be set. To set this variable:

```
setspecies(kineticlawObj, 'S', 'a');
```

- 4 Verify that the species variable is correct.

```
get(kineticlawObj, 'SpeciesVariableNames')
```

MATLAB returns:

```
ans =
```

```
'a'
```

## See Also

`addparameter`, `getspecies`, `setparameter`

**Introduced in R2006a**



# setTable(ScheduleDose,RepeatDose)

Set dosing information from table to dose object

## Syntax

```
setTable(doseObj, tbl)
```

## Description

setTable(doseObj, tbl) sets the dosing data from a table tbl to a dose object doseObj.

## Input Arguments

### doseObj — Dose object

ScheduleDose object | RepeatDose object | array of dose objects

Dose object, specified as a ScheduleDose object or RepeatDose object or array of these objects.

### tbl — Dosing data

table | cell array of tables

Dosing data, specified as a table or cell array of tables. If doseObj is an array of dose objects, then tbl must be a cell array of tables of the same size as doseObj.

If doseObj is a ScheduleDose object, tbl must have 2 or 3 variables (columns) representing dose time, amount, and rate (optional). The variable names (tbl.Properties.VariableNames) must be 'Time', 'Amount', and 'Rate' (optional), respectively.

If doseObj is a RepeatDose object, tbl must have only one row with 4 or 5 variables (columns) representing dose start time, amount, interval, repeat count, and rate (optional). The variable names (tbl.Properties.VariableNames) must be 'StartTime', 'Amount', 'Interval', 'RepeatCount', and 'Rate' (optional),

respectively. The value of each variable can be a numeric scalar or the name of a parameter (if the property is parameterized).

The units of `tbl` variables (`tbl.Properties.VariableUnits`), if any, are copied over to the corresponding property units of `doseObj`.

## Examples

### Set a Table of Dosing Data to a RepeatDose Object

Create a table containing dose start time, amount, interval, repeat count, and rate.

```
StartTime = 5;  
Amount = 500;  
Interval = 1;  
RepeatCount = 3;  
Rate = 1;  
tbl = table(StartTime,Amount,Interval,RepeatCount,Rate);
```

Create a `RepeatDose` object, and set the dosing information from the table.

```
rdose = sbiodose('rdose','repeat');  
setTable(rdose,tbl);
```

### Set a Table of Dosing Data to a ScheduleDose Object

Create a table containing dose time and amount.

```
Time = [1 2 3 4 5]';  
Amount = [10 15 20 25 30]';  
tbl = table(Time,Amount);
```

Create a `ScheduleDose` object, and set the dosing information from the table.

```
sdose = sbiodose('sdose','schedule');  
setTable(sdose,tbl);
```

## Set an Array of Dosing Tables to an Array of Dose Objects

Create a table containing dose time and amount.

```
Time = [1 2 3 4 5]';  
Amount = [10 15 20 25 30]';  
tbl1 = table(Time,Amount);
```

Create a table containing dose start time, amount, interval, repeat count, and rate.

```
StartTime = 5;  
Amount = 500;  
Interval = 1;  
RepeatCount = 3;  
Rate = 1;  
tbl2 = table(StartTime,Amount,Interval,RepeatCount,Rate);
```

Create a cell array of dose tables.

```
tblArray = {tbl1,tbl2};
```

Create ScheduleDose and RepeatDose objects

```
sdose = sbiodose('sdose','schedule');  
rdose = sbiodose('rdose','repeat');  
doseArray = [sdose,rdose];
```

Set the cell array of dose tables to dose objects.

```
setTable(doseArray,tblArray);
```

## See Also

[RepeatDose object](#) | [ScheduleDose object](#) | [getTable](#)

**Introduced in R2014a**

## SimData object

Simulation data storage

### Description

The SimBiology `SimData` object contains simulation data. The output from the `sbiosimulate` function, is stored in the `SimData` object which holds time and state data as well as metadata, such as the types and names for the logged states or the configuration set used during simulation.

You can also store data from multiple simulation runs as an array of `SimData` objects. Thus, the output of `sbioenssemblerun` is an array of `SimData` objects. You can use any `SimData` method on an array of `SimData` objects.

You can access the time, data, and metadata stored in the `SimData` object through the properties in “Property Summary” on page 2-457. Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line.

Methods you can use to query the `SimData` object are listed in “Method Summary” on page 2-457.

### Constructor Summary

<code>sbioenssemblerun</code>	Multiple stochastic ensemble runs of SimBiology model
<code>sbiosimulate</code>	Simulate SimBiology model

## Method Summary

delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
get (any object)	Get object properties
getdata (SimData)	Get data from SimData object array
getsensmatrix (SimData)	Get 3-D sensitivity matrix from SimData array
resample (SimData)	Resample SimData object array onto new time vector
select (SimData)	Select data from SimData object
selectbyname (SimData)	Select data by name from SimData object array
set (any object)	Set object properties

## Property Summary

Data	Store simulation data
DataCount	Numbers of species, parameters, sensitivities
DataInfo	Metadata labels for simulation data
DataNames	Show names in SimData object
ModelName	Name of model simulated
Name	Specify name of object
Notes	HTML text describing SimBiology object
RunInfo	Information about simulation
Time	Simulation time steps or schedule dose times
TimeUnits	Show time units for dosing and simulation
UserData	Specify data to associate with object

## Examples

Return simulation results to a SimData object and plot the results.

```
sbioloadproject('radiodecay', 'm1');
simDataObj = sbiosimulate(m1);
sbioplot(simDataObj)
```

Get simulation data at specific time points using the `resample` method.

```
% Load 'radiodecay' and set the initial amount of species 'x'.
sbioloadproject('radiodecay', 'm1');
x = sbioselect(m1, 'Type', 'species', 'Name', 'x');
x.InitialAmount = 100;
% Change the solver type to a stochastic solver.
cs = m1.getconfigset;
cs.SolverType = 'ssa';
% Simulate the model.
simDataObj = sbiosimulate(m1);
% This result could be misinterpreted as containing fractional molecules.
sbioplot(simDataObj);
title('Simulation Results Before Resampling');
% Resample the data using the zero-order hold method to obtain the correct
% number of molecules at intermediate time steps.
newsimDataObj = resample(simDataObj, linspace(0, 10, 1e4), 'zoh');
sbioplot(newsimDataObj);
title('Simulation Results After Resampling');
```

Initialize a simulation using results from a previous simulation.

```
% Load 'radiodecay'.
sbioloadproject('radiodecay', 'm1');
m1.Species
simDataObj = sbiosimulate(m1);
% Use the Data property to get the states at the final time point.
% Data is an m x n array, where m is the number of time steps in
% the simulation and n is the number of quantities logged.
finaldata = simDataObj.Data(end,:);
% Use the DataInfo property to get names of states.
info = simDataObj.DataInfo;
% Loop through the states (species) and set their initial amounts.
numSpecies = length(info);
for c = 1:numSpecies
    compObj = sbioselect(m1, 'type', 'compartment', 'Name', info{c}.Compartment);
    speciesObj = sbioselect(compObj, 'type', 'species', 'Name', info{c}.Name);
    speciesObj.InitialAmount = finaldata(c);
end
% Verify species initial amounts.
m1.Species
```

## **See Also**

Model object, Parameter object, Reaction object, Root object, Species object

**Introduced in R2007b**

## SimFunction object

Function-like interface to execute SimBiology models

### Description

The `SimFunction` object provides an interface that allows you to execute a SimBiology model like a function and a workflow to perform parameter scans (in parallel if Parallel Computing Toolbox is available), Monte Carlo simulations, and scans with multiple or vectorized doses. Since a `SimFunction` object can be executed like a function handle, you can customize it to integrate SimBiology models with other MATLAB products and other custom analyses (such as visual predictive checks).

Use the `createSimFunction` method to construct the `SimFunction` object. `SimFunction` objects are immutable once created and automatically accelerated at the first function execution.

### Syntax

If you specified any dosing information when you called `createSimFunction` to construct the `SimFunction` object `F`, then `F` has the following syntaxes.

`simdata = F(phi,t_stop,u,t_output)` returns a `SimData` object `simdata` after simulating a SimBiology model using the initial conditions specified in `phi`, simulation stop time, `t_stop`, dosing information, `u`, and output time, `t_output`.

`simdata = F(phi,t_stop,u)` runs simulations using the input arguments `phi`, `t_stop`, and `u`.

If you did *not* specify any dosing information when you called `createSimFunction`, then `F` has the following syntaxes:

`simdata = F(phi,t_stop)` returns a `SimData` object `simdata` after simulating the model using initial conditions specified in `phi`, and simulation stop time, `t_stop`.

`simdata = F(phi,t_stop,[],t_output)` uses the input arguments `phi`, `t_stop`, empty dosed argument `[]`, and `t_output`. You must specify `u`, the dosing information, as



an empty array [] for this signature. When `t_output` is empty and `t_stop` is specified, the simulations report the solver time points until `t_stop`. When `t_output` is specified and `t_stop` is empty, only the time points in `t_output` are reported. When both are specified, the reported time points are the union of solver time points and the time points in `t_output`. If the last `t_output` is greater than the corresponding `t_stop`, then simulation proceeds until the last time point in `t_output`.

`simdata = F(phi,tbl)` uses the input arguments `phi` and `tbl`. Using this signature only lets you specify output times as one of the variables of `tbl`. Any data row in `tbl` where all dependent variable columns having NaN values is ignored.

`[T,Y] = F(_)` returns `T`, a cell array of numeric vector, and `Y`, a cell array of 2-D numeric matrices, using any of the input arguments in the preceding syntaxes.

## Input Arguments

### `phi`

One of the following:

- Empty array [] or empty cell array {}, meaning to perform simulations using the baseline initial values, that is, the values listed in the `Parameter` property of the `SimFunction` object, without altering them.
- Matrix of size *S*-by-*P*, where *S* is the number of simulations to perform and *P* is the number of parameters specified in the `params` argument when you called `createSimFunction` to construct `F`. Each simulation is performed with the parameters specified in the corresponding row of `phi`.
- *S*-by-*V* matrix of variant objects or a cell column vector of length *S*, where each element consists of a row vector of variant objects. *S* is the number of simulations to perform, and *V* is the number of variant objects. These variants are only allowed to modify the `SimFunction` input parameters, that is, model elements that were specified as the `params` input argument when you called `createSimFunction`. In other words, you must specify the variant parameters as the input parameters when you create the `SimFunction` object. Any `SimFunction` input parameters that are not specified in the variants use their baseline initial values.

If, within a row of variants, multiple entries refer to the same model element, the last occurrence is used for simulation.

When `phi` is specified as a 1-by- $P$  or 1-by- $V$  matrix, then all simulations use the same parameters, and the number of simulations is determined from the `t_stop`, `u`, or `t_output` argument in that order. For example, if `phi` and `t_stop` have a single row and `u` is a matrix of size  $N$ -by-*DoseTargets*, the number of simulations is determined as  $N$ .

### **t\_stop**

- Scalar specifying the same stop time for all simulations
- Vector of size  $N$  specifying a stop time for each simulation for all  $N$  simulations

### **u**

- Empty array `[]` to apply no doses during simulation.
- table of dosing information with two or three variables containing `ScheduleDose` data (`ScheduleDose` table), namely, dose time, dose amount, and dose rate (optional). Name the table variables as follows.

```
u.Properties.VariableNames = {'Time', 'Amount', 'Rate'};
```

If `UnitConversion` is on, specify units for each variable. For instance, you can specify units as follows.

```
u.Properties.VariableUnits = {'second', 'molecule', 'molecule/second'};
```

This table can have multiple rows, where each row represents a dose applied to the dose target at a specified dose time with a specified amount and rate if available.

- table with one row and five variables containing `RepeatDose` data (`RepeatDose` table). Dose rate variable is optional. Name the variables as follows.

```
u.Properties.VariableNames = {'StartTime', 'Amount', 'Rate', 'Interval', 'RepeatCount'};
```

If `UnitConversion` is on, specify units for each variable. Units for `'RepeatCount'` variable can be empty `''` or `'dimensionless'`. The unit of the `'Amount'` variable must be dimensionally consistent with that of the target species. For example, if the unit of target species is in an *amount* unit (such as mole or molecule), then the `'Amount'` variable unit must have the same dimension, i.e., its unit must be an *amount* unit and cannot be a *mass* unit (such as gram or kilogram). The unit for the `'Rate'` variable must be dimensionally consistent as well.

```
u.Properties.VariableUnits = {'second', 'molecule', 'molecule/second', 'second', 'dimensionless'};
```

---

**Tip** If you already have a dose object (`ScheduleDose` or `RepeatDose`), you can get this dose table by using the `getTable` method of the object.

---

- Cell array of tables of size 1-by-N, where N is the number of dose targets. Each cell can represent either table as described previously.
- Cell array of tables of size S-by-N, where S is the number of simulations and N is the number of dose targets. Each cell represents a table. S is equal to the number of rows in `phi`.

If `u` is a cell array of tables, then:

- The number of columns (*N*) in the cell array `u` must be equal to the number of elements in the `dosed` input argument on page 2-0 of `createSimFunction`. The order of dose tables must also match the order of dosed species in `createSimFunction`. That is, `SimBiology` assumes one-to-one correspondence between the columns of `u` and dose targets specified in the `dosed` on page 2-0 input argument of `createSimFunction`, meaning the doses (dose tables) in the first column of `u` are applied to the first dose target in `dosed` on page 2-0 and so on.
- The *i*th dose for the *j*th dose target is ignored if `u{i,j} = []`.
- If the *i*th dose is not parameterized, `u{i,j}` can be `[]` or either type of table (the `ScheduleDose` or `RepeatDose` table).
- If the *i*th dose is parameterized, `u{i,j}` must be `[]` or a `RepeatDose` table with one row and a column for each property (`StartTime`, `Amount`, `Rate`, `Interval`, `RepeatCount`) that is not parameterized. It is not required to create a column for a dose property that is parameterized. If all of the properties are parameterized, you can pass in a table with one row and no columns to specify the parameterized dose is applied during simulations. To create such table, use `table.empty(1,0)`.

### **t\_output**

- Vector of monotonically increasing output times that is applied to all simulations
- Cell array containing a single time vector that is applied to all simulations
- Cell array of vectors representing output times. The *i*th cell element provides the output times for the *i*th simulation. The number of elements in the cell array must match the number of rows (simulations) in `phi`.

### **tbl**

`table` or `dataset` that has time and dosing information such as group labels, independent variable, dependent variable(s), amount(s), and rate(s). You must name the

variables of the table or data set as

'GROUP', 'TIME', 'DEPENDENTVAR1', 'DEPENDENTVAR2', ..., 'AMOUNT1', 'RATE1', 'AMOUNT2', 'RATE2', ... The rate variable is optional for each dose.

If the `dosed` argument was empty when creating `F`, then amount- and rate-related variables are not required. If it is not empty, the number of amount and rate variables must match the number of dosed targets or species in `dosed`. The number of dependent variables must match the number of columns in `phi`.

If `UnitConversion` is on, specify a unit for each variable. The unit of 'Amount' variable must be dimensionally consistent with that of the target species. See the description of the input argument `u` for details.

## Output Arguments

### `simdata`

Array of `SimData` objects that contains results from executing the `SimFunction` `F`. The number of elements in the `simdata` array is the same as the number of rows in `phi`. The number of columns in each element of the `simdata` array, that is, `simdata(i).Data`, is equal to the number of elements in the observed cell array which was specified when creating `F`.

### `T`

Cell array containing a numeric vector of size  $S \times 1$ .  $S$  is the number of simulations. The  $i$ th element of `T` contains the time point from the  $i$ th simulation.

### `Y`

Cell array of 2-D numeric matrices. The  $i$ th element of `Y` contains data from the  $i$ th simulation. The number of rows in `T{i}` is equal to the number of rows in `Y{i}`.

## Constructor Summary

`createSimFunction(model)`

Create `SimFunction` object

## Method Summary

`accelerate(SimFunction)` Prepare SimFunction object for accelerated simulations  
`isaccelerated(SimFunction)` Determine if SimFunction object is accelerated

## Property Summary

Parameters	<p>table with variables named:</p> <ul style="list-style-type: none"> <li>• 'Name'</li> <li>• 'Value'</li> <li>• 'Type'</li> <li>• 'Units' (only if <code>UnitConversion</code> is turned on)</li> </ul> <p>The table contains information about model quantities (species, compartments, or parameters) that define the inputs of a <code>SimFunction</code> object. For instance, this table can contain parameters or species whose values are being scanned by the <code>SimFunction</code> object. This property is read only.</p>
Observables	<p>table with variables named:</p> <ul style="list-style-type: none"> <li>• 'Name'</li> <li>• 'Type'</li> <li>• 'Units' (only if <code>UnitConversion</code> is turned on)</li> </ul> <p>This table contains information about model quantities (species, compartments, or parameters) that define the output of a <code>SimFunction</code> object. This property is read only.</p>

Dosed	<p>table containing dosing information with variables named:</p> <ul style="list-style-type: none"> <li>• 'TargetName'</li> <li>• 'TargetDimension' (only if UnitConversion is turned on)</li> </ul> <p>In addition, the table also contains variables for each property that is parameterized. For each parameterized property, two variables are added to this table. The first variable has the same name as the property name and the value is the name of the specified parameter. The second variable has the property name suffixed by <i>Value</i> (<i>PropertyNameValue</i>), and the value is the default value of the parameter. If the <i>UnitConversion</i> is on, the unit column is also added with the name <i>PropertyNameUnits</i>.</p> <p>Suppose the <i>Amount</i> property of a repeat dose targeting the <i>Drug</i> species is parameterized by setting it to a model parameter called <i>AmountParam</i> with the value of 10 milligram, and <i>UnitConversion</i> is on. The <i>Dosed</i> table contains the following variables:</p> <table border="1" data-bbox="463 850 1322 1032"> <thead> <tr> <th>TargetName</th> <th>TargetDimension</th> <th>Amount</th> <th>AmountValue</th> <th>AmountUnits</th> </tr> </thead> <tbody> <tr> <td>'Drug'</td> <td>'Mass (e.g., gram)'</td> <td>'AmountParam'</td> <td>10</td> <td>'milligram'</td> </tr> </tbody> </table>	TargetName	TargetDimension	Amount	AmountValue	AmountUnits	'Drug'	'Mass (e.g., gram)'	'AmountParam'	10	'milligram'
TargetName	TargetDimension	Amount	AmountValue	AmountUnits							
'Drug'	'Mass (e.g., gram)'	'AmountParam'	10	'milligram'							
UseParallel	<p>Logical. If true and Parallel Computing Toolbox is available, SimFunction is executed in parallel. This property is read-only.</p>										

UnitConversion	<p>Logical. If true:</p> <ul style="list-style-type: none"> <li>• During the execution of the <code>SimFunction</code> object, <code>phi</code> is assumed to be in the same units as units for corresponding model quantities specified in the <code>params</code> argument when the object was created using the <code>createSimFunction</code> method.</li> <li>• Time (<code>t_output</code> or <code>t_stop</code>) is assumed to be in the same unit as the <code>TimeUnits</code> property of the active <code>configset</code> object of the <code>SimBiology</code> model from which <code>F</code> was created.</li> <li>• Variables of dose tables (<code>u</code>) must have units specified by setting <code>u.Properties.VariableUnits</code> to a cell array of appropriate units. The dimension of the dose target such as an amount (molecule, mole, etc.) or mass (gram, kilogram, etc.), is stored on the <code>Dosed</code> property of <code>F</code>.</li> <li>• The simulation result is in the same units as those specified on the corresponding quantities in the <code>SimBiology</code> model from which <code>F</code> was created.</li> </ul> <p>This property is read only.</p>
DependentFiles	<p>Cell array of character vectors containing the names of files that the model depends on. This property is used for deployment. This property is read only.</p>

## Examples

### Simulate Model of Glucose-Insulin Response with Different Initial Conditions

This example shows how to simulate the glucose-insulin responses for the normal and diabetic subjects.

Load the model of glucose-insulin response. For details about the model, see the **Background** section in “Simulating the Glucose-Insulin Response”.

```
sbioloadproject('insulindemo', 'm1')
```

The model contains different initial conditions stored in various variants.

```
variants = getvariant(m1);
```

Get the initial conditions for the type 2 diabetic patient.

```
type2 = variants(1)
```

```
SimBiology Variant - Type 2 diabetic (inactive)
```

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	Plasma Volume ...	Value	1.49
2	parameter	k1	Value	0.042
3	parameter	k2	Value	0.071
4	parameter	Plasma Volume ...	Value	0.04
5	parameter	m1	Value	0.379
6	parameter	m2	Value	0.673
7	parameter	m4	Value	0.269
8	parameter	m5	Value	0.0526
9	parameter	m6	Value	0.8118
10	parameter	Hepatic Extrac...	Value	0.6
11	parameter	kmax	Value	0.0465
12	parameter	kmin	Value	0.0076
13	parameter	kabs	Value	0.023
14	parameter	kgri	Value	0.0465
15	parameter	f	Value	0.9
16	parameter	a	Value	6e-05
17	parameter	b	Value	0.68
18	parameter	c	Value	0.00023
19	parameter	d	Value	0.09
20	parameter	Stomach Glu Af...	Value	125
21	parameter	kp1	Value	3.09
22	parameter	kp2	Value	0.0007
23	parameter	kp3	Value	0.005
24	parameter	kp4	Value	0.0786
25	parameter	ki	Value	0.0066
26	parameter	[Ins Ind Glu U...	Value	1
27	parameter	Vm0	Value	4.65
28	parameter	Vmx	Value	0.034
29	parameter	Km	Value	466.21
30	parameter	p2U	Value	0.084
31	parameter	K	Value	0.99
32	parameter	alpha	Value	0.013
33	parameter	beta	Value	0.05
34	parameter	gamma	Value	0.5
35	parameter	ke1	Value	0.0007
36	parameter	ke2	Value	269
37	parameter	Basal Plasma G...	Value	164.18
38	parameter	Basal Plasma I...	Value	54.81



Suppress an informational warning that is issued during simulations.

```
warnSettings = warning('off','SimBiology:DimAnalysisNotDone_MatlabFcn_Dimensionless');
```

Create SimFunction objects to simulate the glucose-insulin response for the normal and diabetic subjects.

- Specify an empty array {} for the second input argument to denote that the model will be simulated using the base parameter values (that is, no parameter scanning will be performed).
- Specify the plasma glucose and insulin concentrations as responses (outputs of the function to be plotted).
- Specify the species Dose as the dosed species. This species represents the initial concentration of glucose at the start of the simulation.

```
normSim = createSimFunction(m1, {}, ...
    {'[Plasma Glu Conc]', '[Plasma Ins Conc]'}, 'Dose')
```

```
normSim =
SimFunction
```

Parameters:

Observables:

Name	Type	Units
'[Plasma Glu Conc]'	'species'	'milligram/deciliter'
'[Plasma Ins Conc]'	'species'	'picomole/liter'

Dosed:

TargetName	TargetDimension
'Dose'	'Mass (e.g., gram)'

For the diabetic patient, specify the initial conditions using the variant type2.

```
diabSim = createSimFunction(m1, {}, ...
    {'[Plasma Glu Conc]', '[Plasma Ins Conc]'}, 'Dose', type2)
```

```
diabSim =  
SimFunction
```

Parameters:

Observables:

Name	Type	Units
'[Plasma Glu Conc]'	'species'	'milligram/deciliter'
'[Plasma Ins Conc]'	'species'	'picomole/liter'

Dosed:

TargetName	TargetDimension
'Dose'	'Mass (e.g., gram)'

Select a dose that represents a single meal of 78 grams of glucose at the start of the simulation.

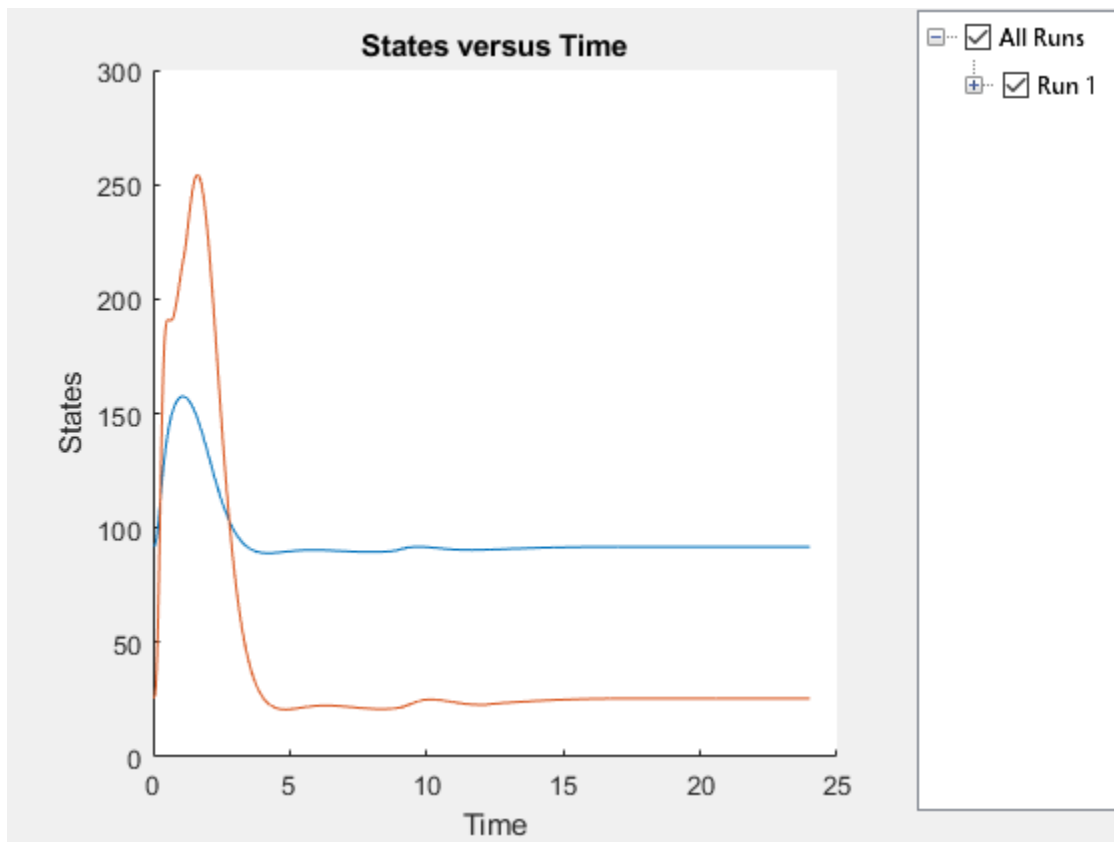
```
singleMeal = sbioselect(m1, 'Name', 'Single Meal');
```

Convert the dosing information to the table format.

```
mealTable = getTable(singleMeal);
```

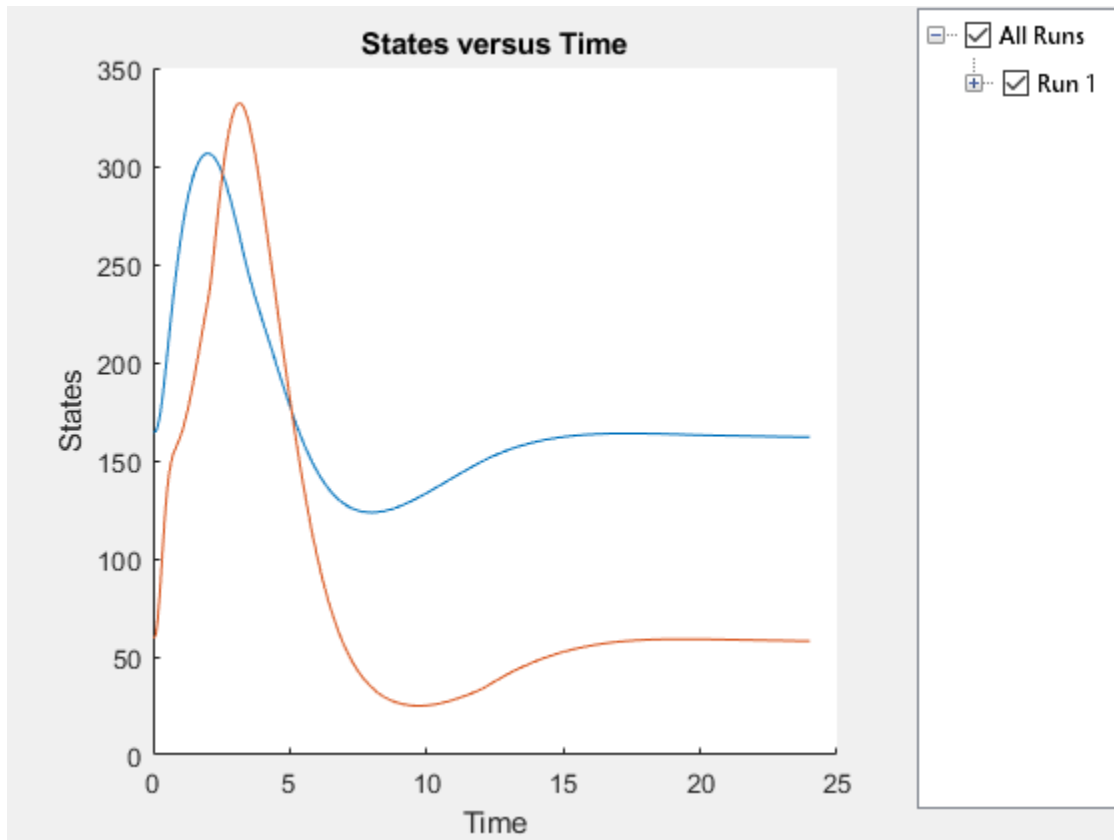
Simulate the glucose-insulin response for a normal subject for 24 hours.

```
sbioplot(normSim([], 24, mealTable));
```



Simulate the glucose-insulin response for a diabetic subject for 24 hours.

```
sbioplot(diabSim([],24,mealTable));
```



### Perform a scan using variants

Suppose you want to perform a parameter scan using an array of variants that contain different initial conditions for different insulin impairments. For example, the model `m1` has variants that correspond to the low insulin sensitivity and high insulin sensitivity. You can simulate the model for both conditions via a single call to the `SimFunction` object.

Select the variants to scan.

```
varToScan = sbioselect(m1, 'Name', ...
    {'Low insulin sensitivity', 'High insulin sensitivity'});
```

Check which model parameters are being stored in each variant.

```
varToScan(1)
```

SimBiology Variant - Low insulin sensitivity (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	Vmx	Value	0.0235
2	parameter	kp3	Value	0.0045

varToScan(2)

SimBiology Variant - High insulin sensitivity (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	parameter	Vmx	Value	0.094
2	parameter	kp3	Value	0.018

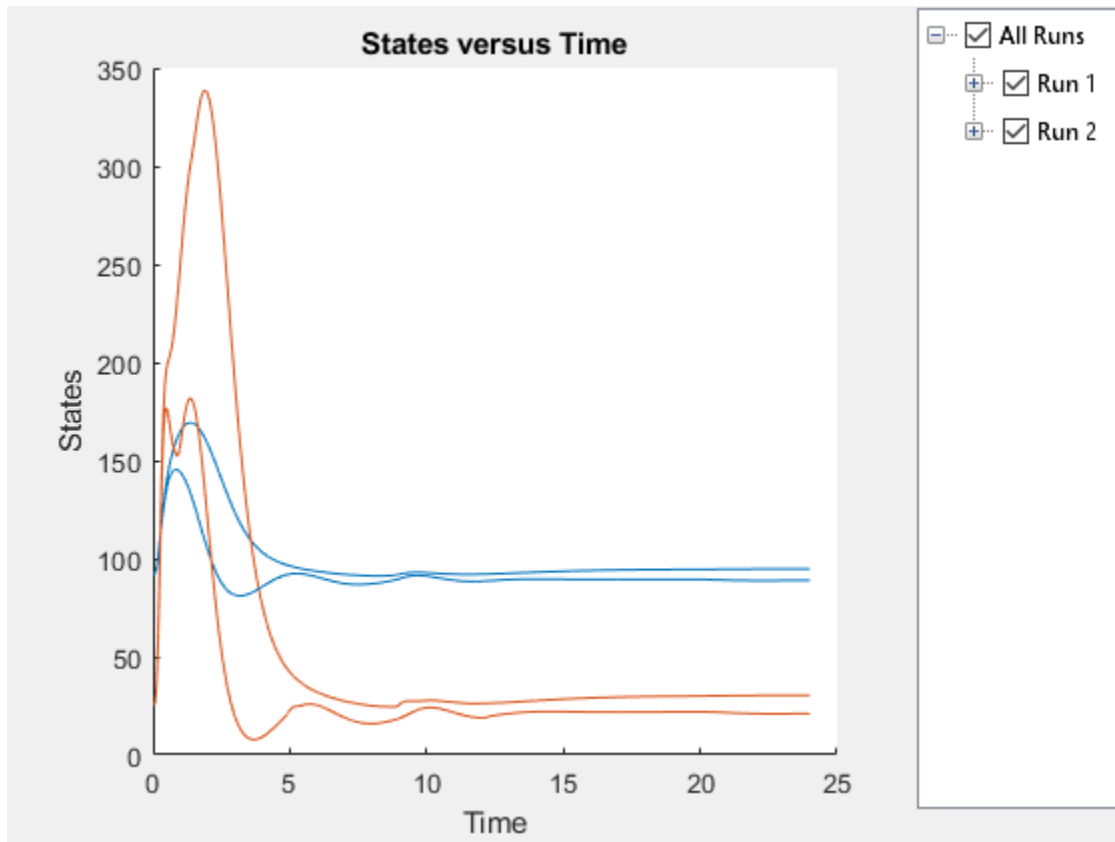
Both variants store alternate values for Vmx and kp3 parameters. You need to specify them as input parameters when you create a SimFunction object.

Create a SimFunction object to scan the variants.

```
variantScan = createSimFunction(m1, {'Vmx', 'kp3'}, ...
    {'[Plasma Glu Conc]', '[Plasma Ins Conc]'}, 'Dose');
```

Simulate the model and plot the results. Run 1 include simulation results for the low insulin sensitivity and Run 2 for the high insulin sensitivity.

```
sbioplot(variantScan(varToScan,24,mealTable));
```



Low insulin sensitivity lead to increased and prolonged plasma glucose concentration.

Restore warning settings.

```
warning(warnSettings);
```

### Scan Initial Amounts of a Species from a Radioactive Decay Model

This example shows how to scan initial amounts of a species from a radioactive decay

model with the first-order reaction  $\frac{dz}{dt} = c \cdot x$ , where x and z are species and c is the forward rate constant.

Load the sample project containing the radiodecay model m1.

```
sbioloadproject radiodecay;
```

Create a SimFunction object f to scan initial amounts of species x.

```
f = createSimFunction(m1,{'x'},{'x','z'},[])
```

```
f =  
SimFunction
```

Parameters:

Name	Value	Type	Units
'x'	1000	'species'	'molecule'

Observables:

Name	Type	Units
'x'	'species'	'molecule'
'z'	'species'	'molecule'

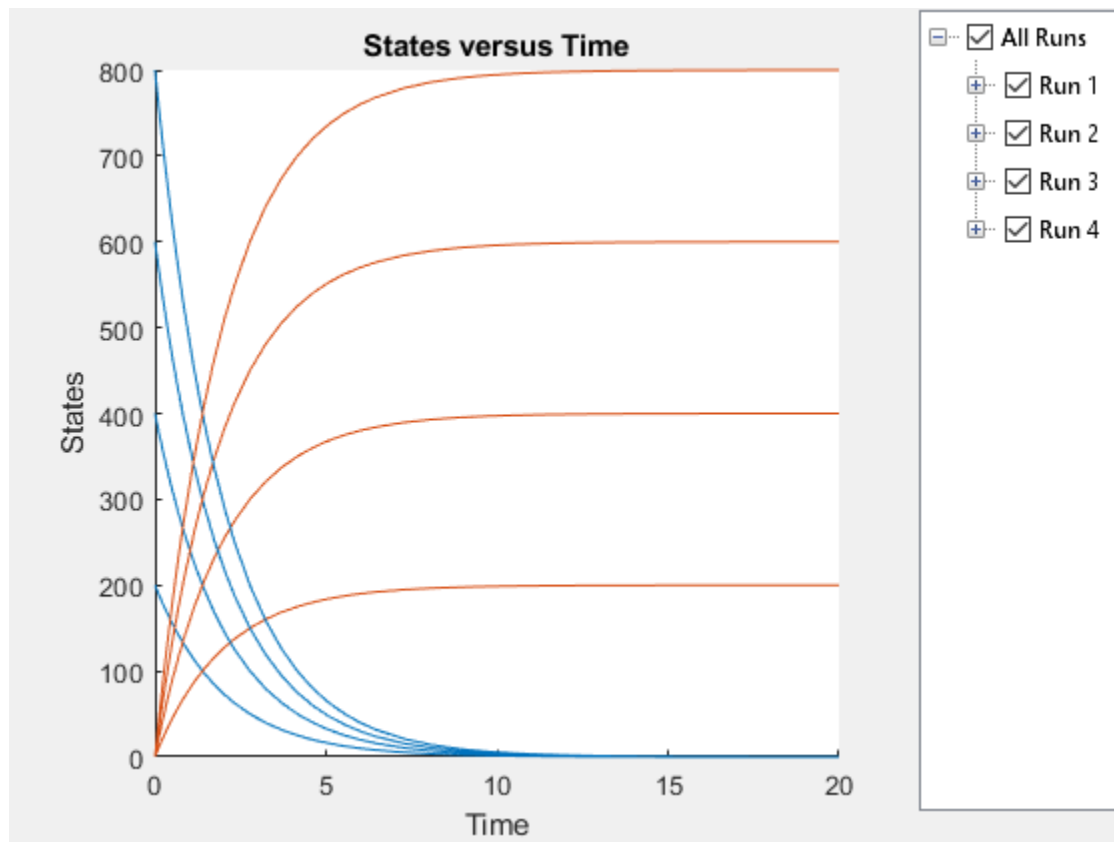
Dosed: None

Define four different initial amounts of species x for scanning. The number of rows indicates the total number of simulations, and each simulation uses the parameter value specified in each row of the vector.

```
phi = [200; 400; 600; 800];
```

Run simulations until the stop time is 20 and plot the simulation results.

```
sbioplot(f(phi, 20));
```



### Simulate a Model and Scan Parameters with Doses

This example shows how to simulate and scan a parameter of a radiodecay model while a species is being dosed.

Load the sample project containing the radiodecay model `m1`.

```
sbioloadproject radiodecay;
```

Create a `SimFunction` object `f` specifying parameter `Reaction1.c` to be scanned and species `x` as a dosed target.



```
f = createSimFunction(m1,{'Reaction1.c'},{'x','z'},{'x'});
```

Define a scalar dose of amount 200 molecules given at three time points (5, 10, and 15 seconds).

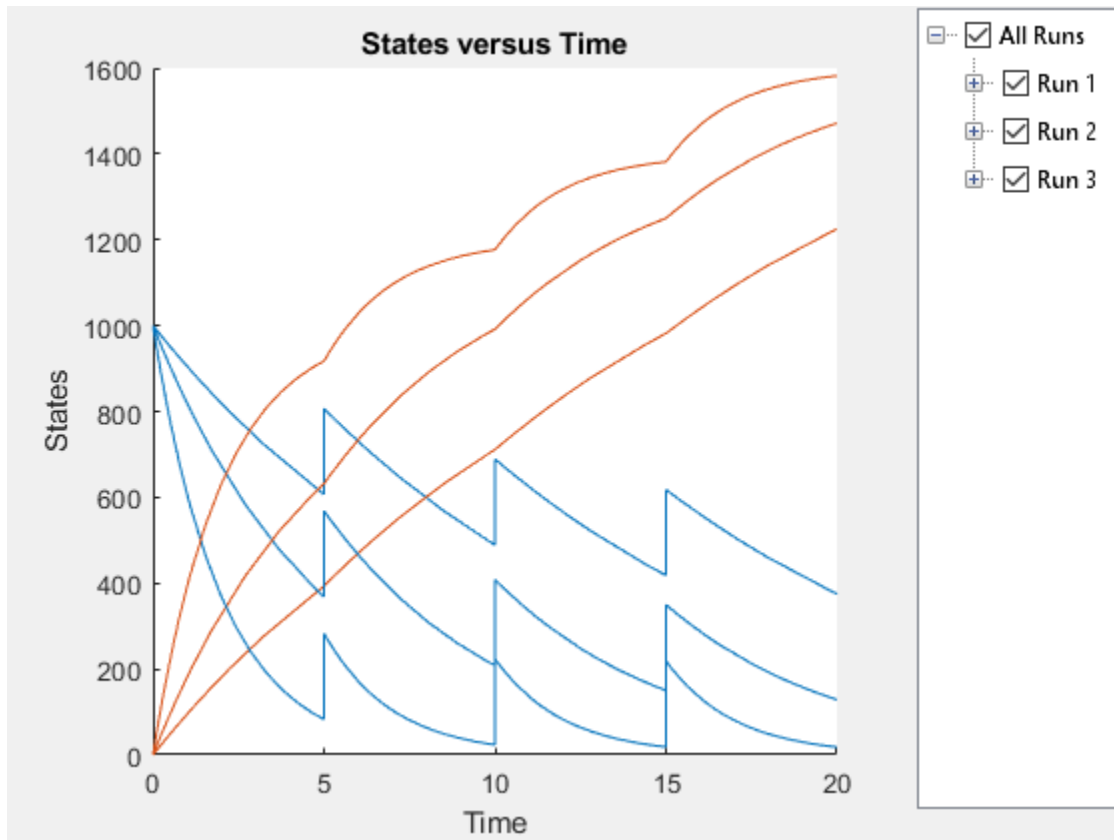
```
dosetime = [5 10 15];  
dose = [200 200 200];  
u = table(dosetime', dose');  
u.Properties.VariableNames = {'Time','Amount'};  
u.Properties.VariableUnits = {'second','molecule'};
```

Define the parameter values for Reaction1.c to scan.

```
phi = [0.1 0.2 0.5]';
```

Simulate the model for 20 seconds and plot the results.

```
sbioplot(f(phi,20,u));
```

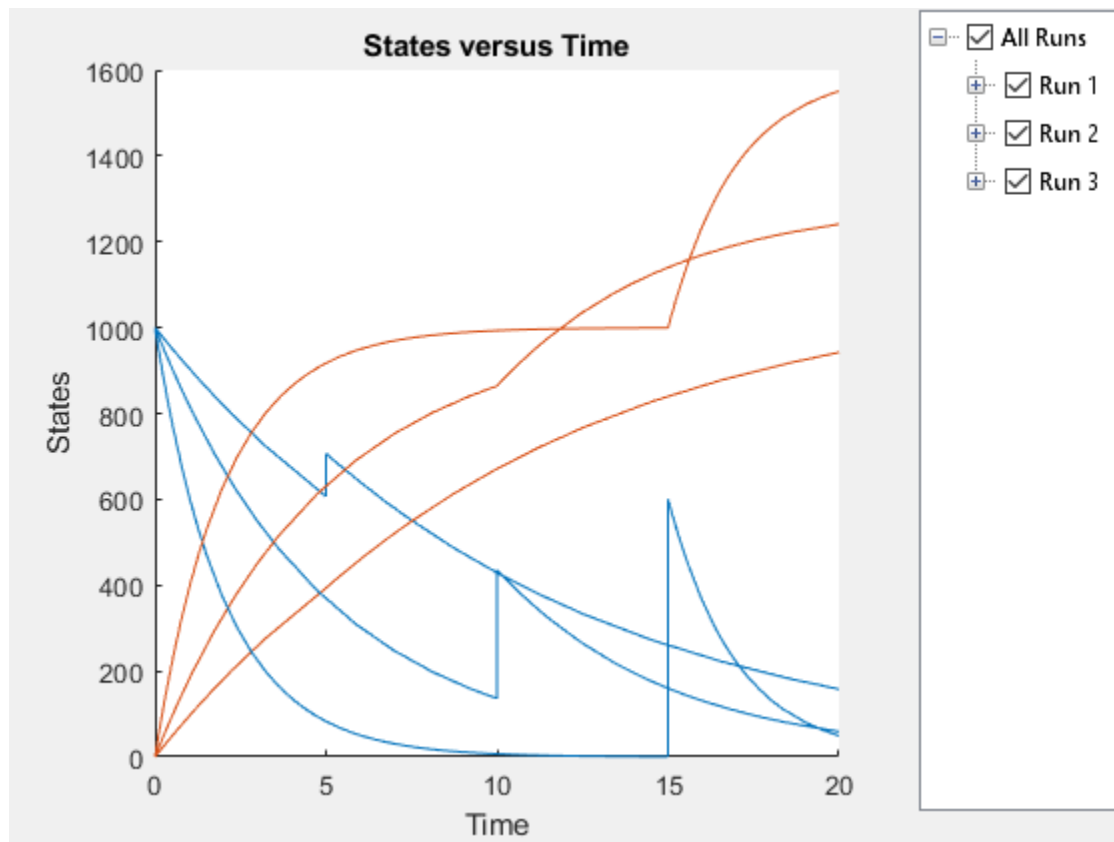


You can also specify different dose amounts at different times.

```
d1 = table(5,100);
d1.Properties.VariableNames = {'Time', 'Amount'};
d1.Properties.VariableUnits = {'second', 'molecule'};
d2 = table(10,300);
d2.Properties.VariableNames = {'Time', 'Amount'};
d2.Properties.VariableUnits = {'second', 'molecule'};
d3 = table(15,600);
d3.Properties.VariableNames = {'Time', 'Amount'};
d3.Properties.VariableUnits = {'second', 'molecule'};
```

Simulate the model using these doses and plot the results.

```
sbioplot(f(phi,20,{d1;d2;d3}));
```



You can also define a cell array of dose tables.

```

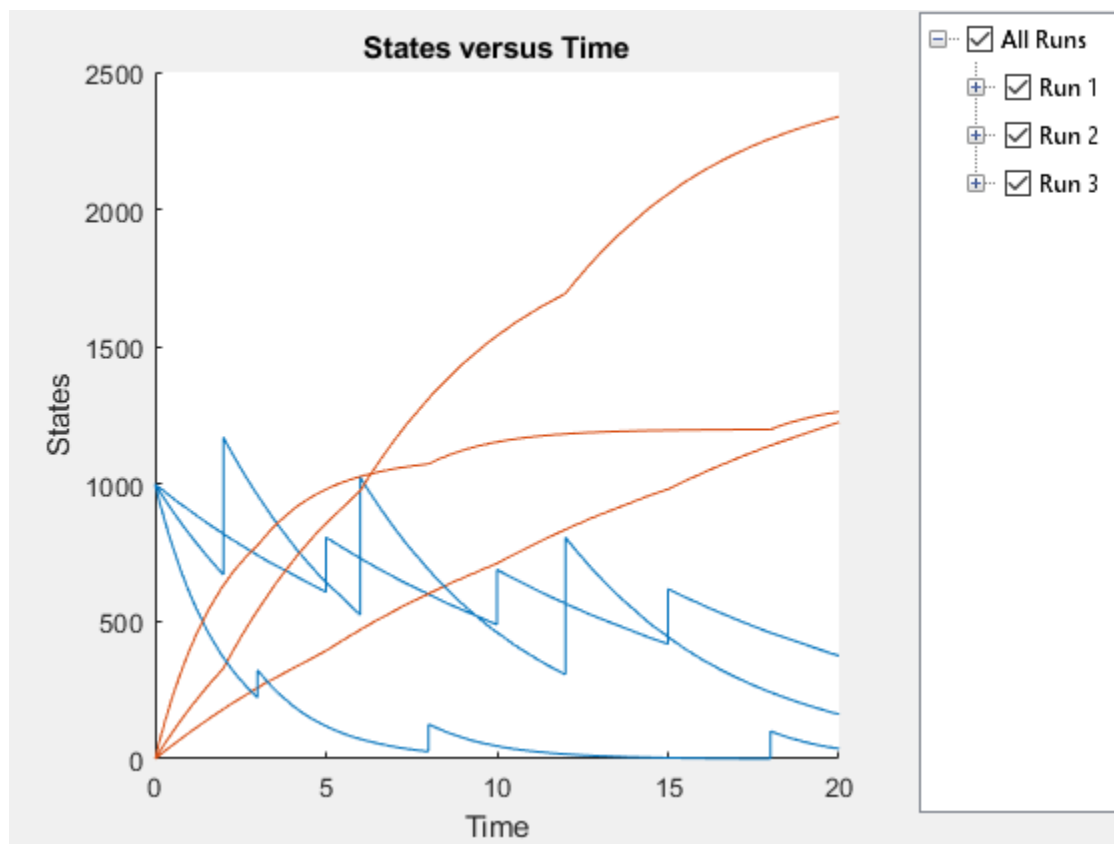
u = cell(3,1);
dosetime = [5 10 15];
dose = [200 200 200];
u{1} = table(dosetime',dose');
u{1}.Properties.VariableNames = {'Time','Amount'};
u{1}.Properties.VariableUnits = {'second','molecule'};
dosetime2 = [2 6 12];
dose2 = [500 500 500];
u{2} = table(dosetime2',dose2');
u{2}.Properties.VariableNames = {'Time','Amount'};
u{2}.Properties.VariableUnits = {'second','molecule'};
dosetime3 = [3 8 18];

```

```
dose3 = [100 100 100];  
u{3} = table(dosetime3', dose3');  
u{3}.Properties.VariableNames = {'Time', 'Amount'};  
u{3}.Properties.VariableUnits = {'second', 'molecule'};
```

Simulate the model using the dose tables and plot results.

```
sbioplot(f(phi,20,u));
```



## References

- [1] Gillespie, D.T. (1977). Exact Stochastic Simulation of Coupled Chemical Reactions. *The Journal of Physical Chemistry*. 81(25), 2340–2361.

## See Also

RepeatDose object | ScheduleDose object | SimFunctionSensitivity object  
| createSimFunction | sbiosampleerror | sbiosampleparameters

## Topics

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”  
“Model Simulation”

**Introduced in R2014a**

## SimFunctionSensitivity object

SimFunctionSensitivity object, subclass of SimFunction object

### Description

The SimFunctionSensitivity object is a subclass of SimFunction object. It allows you to compute sensitivity.

### Syntax

The SimFunctionSensitivity object shares all syntaxes of the SimFunction object. It has the following additional syntax.

[T,Y,SensMatrix] = F( \_\_\_ ) returns T, a cell array of numeric vector, Y, a cell array of 2-D numeric matrices, and SensMatrix, a cell array of 3-D numeric matrix containing calculated sensitivities of model quantities. SensMatrix contains a matrix of size *TimePoints* x *Outputs* x *Inputs*. *TimePoints* is the total number of time points, *Outputs* is the total number of output factors, and *Inputs* is the total number of input factors.

If you specify a single output argument, the object returns an SimData object or array of SimData objects with sensitivity information.

### Property Summary

The SimFunctionSensitivity object shares all properties of the SimFunction object. It has the following additional properties.

SensitivityOutputs table with variables named:

- 'Name'
- 'Type'
- 'Units' (only if UnitConversion is turned on)

This table contains information about model quantities (species or parameters) for which you want to compute the sensitivities. Sensitivity output factors are the numerators of time-dependent derivatives described in “Sensitivity Calculation”. This property is read only.

SensitivityInputs table with variables named:

- 'Name'
- 'Type'
- 'Units' (only if UnitConversion is turned on)

This table contains information about model quantities (species, compartments, or parameters) with respect to which you want to compute the sensitivities. Sensitivity input factors are the denominators of time-dependent derivatives described in “Sensitivity Calculation”. This property is read only.

**SensitivityNormalization** Character vector specifying the normalization method for calculated sensitivities. The following examples show how sensitivities of a species  $x$  with respect to a parameter  $k$  are calculated for each normalization type.

- 'None' — No normalization.

$$\frac{\partial x(t)}{\partial k}$$

- 'Half' — Normalization relative to the numerator only.

$$\left(\frac{1}{x(t)}\right)\left(\frac{\partial x(t)}{\partial k}\right)$$

- 'Full' — Full dedimensionalization

$$\left(\frac{k}{x(t)}\right)\left(\frac{\partial x(t)}{\partial k}\right)$$

## Examples

### Calculate Sensitivities Using SimFunctionSensitivity Object

This example shows how to calculate sensitivities of some species in the Lotka-Volterra model using the SimFunctionSensitivity object.

Load the sample project.

```
sbioloadproject lotka;
```

Define the input parameters.

```
params = {'Reaction1.c1', 'Reaction2.c2'};
```

Define the observed species, which are the outputs of simulation.

```
observables = {'y1', 'y2'};
```



Create a `SimFunctionSensitivity` object. Set the sensitivity output factors to all species (`y1` and `y2`) specified in the `observables` argument and input factors to those in the `params` argument (`c1` and `c2`) by using the keyword `'all'`.

```
f = createSimFunction(m1,params,observables,[],'SensitivityOutputs','all','Sensitivity');
```

```
f =  
SimFunction
```

Parameters:

Name	Value	Type
'Reaction1.c1'	10	'parameter'
'Reaction2.c2'	0.01	'parameter'

Observables:

Name	Type
'y1'	'species'
'y2'	'species'

Dosed: None

Sensitivity Input Factors:

Name	Type
'Reaction1.c1'	'parameter'
'Reaction2.c2'	'parameter'

Sensitivity Output Factors:

Name	Type
'y1'	'species'
'y2'	'species'

Sensitivity Normalization:

Full

Calculate sensitivities by executing the object with `c1` and `c2` set to 10 and 0.1 respectively. Set the output times from 1 to 10. `t` contains time points, `y` contains simulation data, and `sensMatrix` is the sensitivity matrix containing sensitivities of `y1` and `y2` with respect to `c1` and `c2`.

```
[t,y,sensMatrix] = f([10,0.1],[],[],1:10);
```

Retrieve the sensitivity information at simulation time = 5.

```
temp = sensMatrix{:};  
sensMatrix2 = temp(t{:}==5,,:);  
sensMatrix2 = squeeze(sensMatrix2)
```

```
sensMatrix2 = 2×2
```

```
    37.6987    -6.8447  
   -40.2791     5.8225
```

The rows of `sensMatrix2` represent output factors (`y1` and `y2`). The columns represent the input factors (`c1` and `c2`).

$$\mathit{sensMatrix2} = \begin{bmatrix} \frac{\partial y1}{\partial c1} & \frac{\partial y1}{\partial c2} \\ \frac{\partial y2}{\partial c1} & \frac{\partial y2}{\partial c2} \end{bmatrix}$$

Set the stop time to 15, without specifying the output times. In this case, the output times are the solver time points by default.

```
sd = f([10,0.1],15);
```

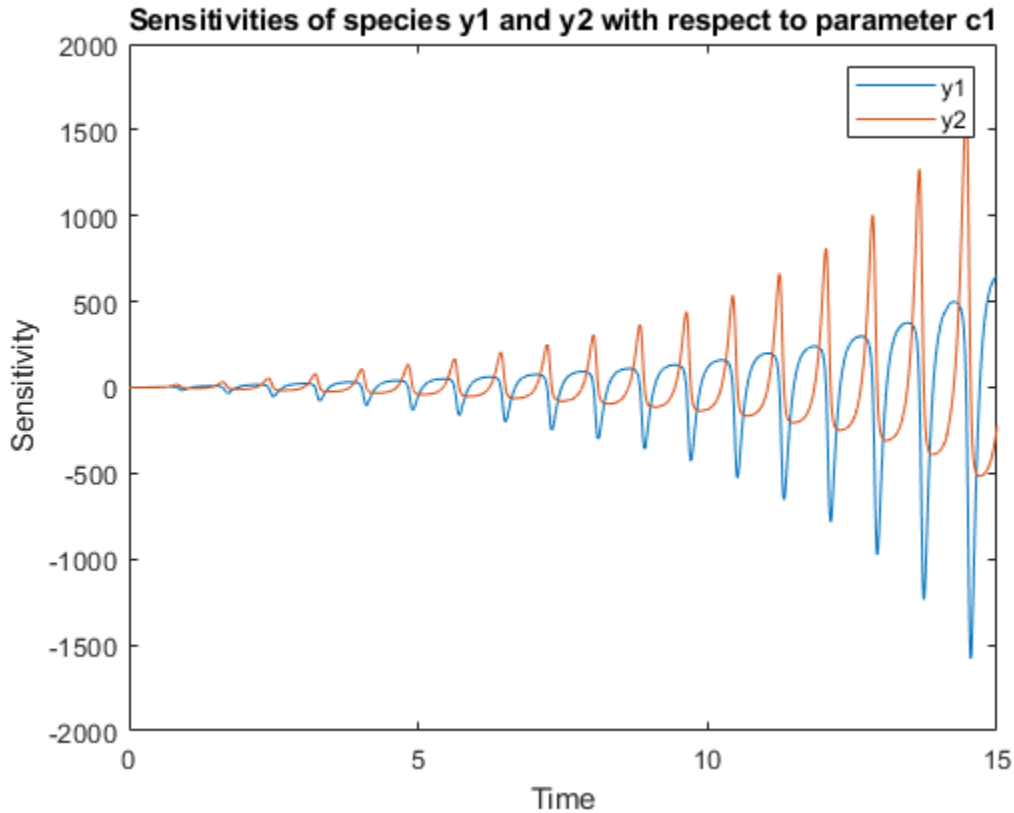
Retrieve the calculated sensitivities from the `SimData` object `sd`.

```
[t,y,outputs,inputs] = getsensmatrix(sd);
```

Plot the sensitivities of species `y1` and `y2` with respect to `c1`.

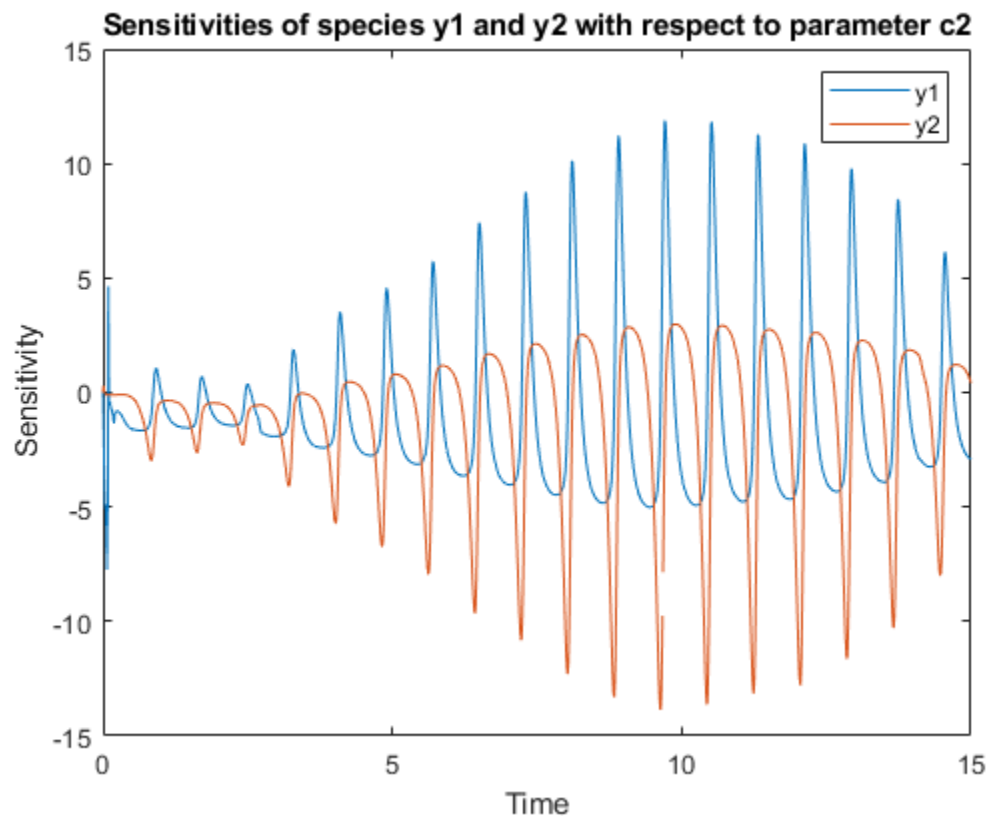
```
figure;  
plot(t,y(:, :, 1));
```

```
legend(outputs);  
title('Sensitivities of species y1 and y2 with respect to parameter c1');  
xlabel('Time');  
ylabel('Sensitivity');
```



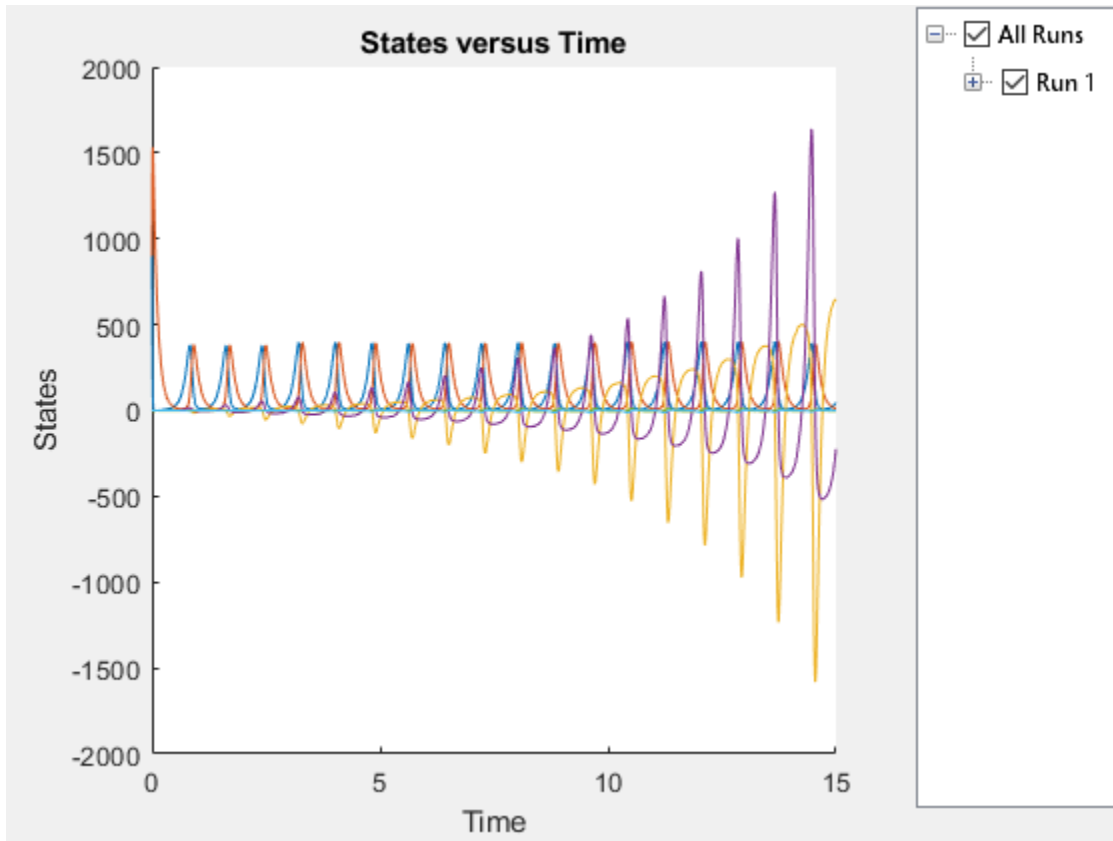
Plot the sensitivities of species y1 and y2 with respect to c2.

```
figure;  
plot(t,y(:,:,2));  
legend(outputs);  
title('Sensitivities of species y1 and y2 with respect to parameter c2');  
xlabel('Time');  
ylabel('Sensitivity');
```



Alternatively, you can use `sbioplot`. Expand Run1 to select which simulation or sensitivity data to display.

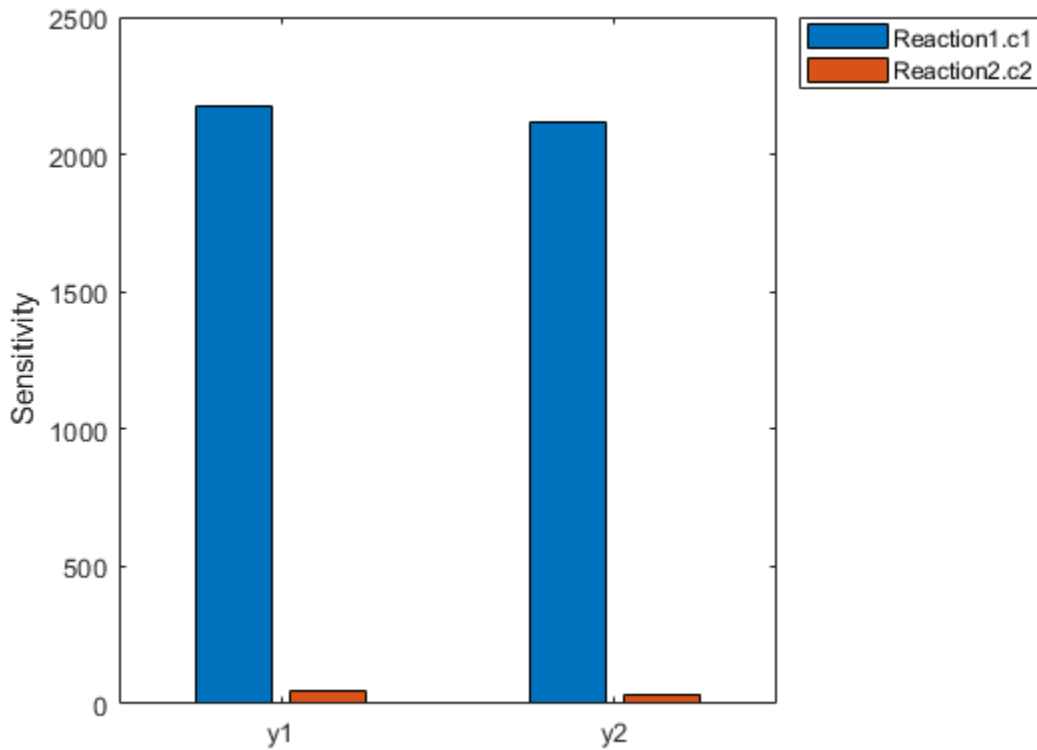
```
sbioplot(sd);
```



You can also plot the sensitivity matrix using the time integral for the calculated sensitivities of  $y_1$  and  $y_2$ . The plot indicates  $y_1$  and  $y_2$  are more sensitive to the parameter  $c_1$  than  $c_2$ .

```
[~, in, out] = size(y);
result = zeros(in, out);
for i = 1:in
    for j = 1:out
        result(i,j) = trapz(t(:),abs(y(:,i,j)));
    end
end
figure;
hbar = bar(result);
haxes = hbar(1).Parent;
```

```
haxes.XTick = 1:length(outputs);  
haxes.XTickLabel = outputs;  
legend(inputs, 'Location', 'NorthEastOutside');  
ylabel('Sensitivity');
```



### See Also

[SimFunction object](#) | [createSimFunction](#) | [sbiosampleerror](#) | [sbiosampleparameters](#)

### Topics

“Sensitivity Calculation”

**Introduced in R2015a**

# simulate

**Class:** SimBiology.export.Model

Simulate exported SimBiology model

## Syntax

```
[t,x,names] = simulate(model)
[t,x,names] = simulate(model,initialValues)
[t,x,names] = simulate(model,initialValues,doses)
simDataObj = simulate( ___ )
```

## Description

`[t,x,names] = simulate(model)` simulates a model, using the default initial values specified by `model.InitialValues` (which are always equal to the `InitialValue` property on the corresponding `ValueInfo` object). `simulate` returns:

- `t`, time samples.
- `x`, simulation data that contain variation in the quantity of states over time.
- `names`, column labels of simulation data `x`.

You can set additional simulation options using the property `SimBiology.export.Model.SimulationOptions`.

`[t,x,names] = simulate(model,initialValues)` simulates a model, using the values specified in `initialValues` as the initial values of the simulation.

`[t,x,names] = simulate(model,initialValues,doses)` simulates the model, using the specified initial values and doses.

`simDataObj = simulate( ___ )` returns simulation data in a `SimData` object `simDataObj` using any of the input arguments in the previous syntaxes. The `simDataObj` contains time and state data, as well as metadata, such as the types and names for the reported states. You can access the time, data, and names stores in



`simDataObj` using the properties `simDataObj.Time`, `simDataObj.Data`, and `simDataObj.DataNames`, respectively.

## Input Arguments

### **model**

`SimBiology.export.Model` object.

### **initialValues**

Vector of values for `simulate` to use as the initial values of the simulation. `initialValues` must have the same number of elements as `model.InitialValues`.

**Default:** Values specified in `model.InitialValues`.

### **doses**

Vector of dose objects specifying the doses used for simulation. The input dose objects must be a subset of the doses in the exported model, as returned by `getdose`.

**Default:** All dose objects in the exported model.

## Output Arguments

### **t**

$n$ -by-1 vector of time samples from the simulation, where  $n$  is the number of time samples.

### **x**

$n$ -by- $m$  matrix of simulation data, where  $n$  is the number of time samples and  $m$  is the number of states logged during the simulation. Each column of `x` describes the variation in the quantity of a state over time.

### **names**

$m$ -by-1 cell array of character vectors with names labeling the rows and columns of `x`, respectively.

**simDataObj**

SimData object containing simulation time and state data, as well as metadata, such as the types and names for the reported states.

## Examples

**Simulate an Exported SimBiology Model**

Load a sample SimBiology model object, and select the species y1 and y2 for simulation.

```
modelObj = sbmlimport('lotka');  
modelObj.getConfigset.RuntimeOptions.StatesToLog = ...  
    sbioselect(modelObj, 'Name', {'y1', 'y2'});
```

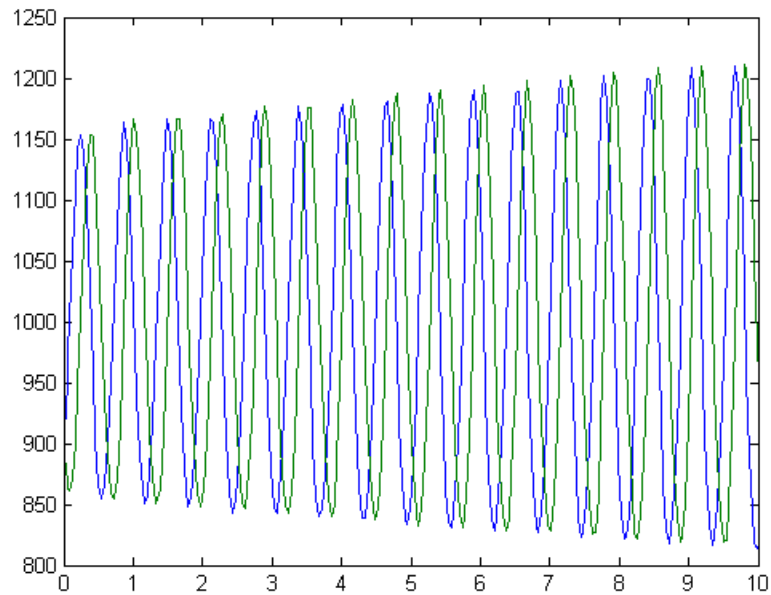
Export the model object.

```
em = export(modelObj);
```

Simulate the exported model.

```
[t,y] = simulate(em);
```

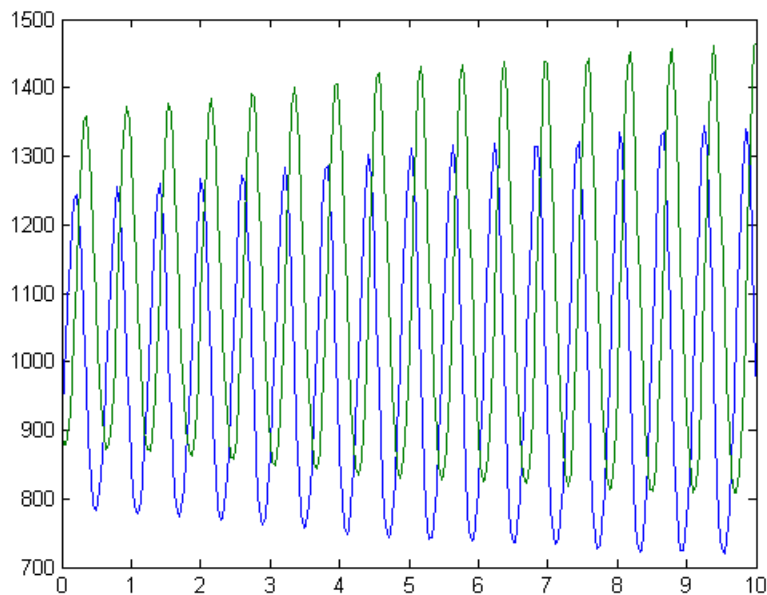
```
figure()  
plot(t,y)
```



Modify the initial conditions, and simulate again.

```
xIndex = em.getIndex('x');  
em.InitialValues(xIndex) = em.InitialValues(xIndex)*1.1;  
[t,y] = simulate(em);
```

```
figure()  
plot(t,y)
```



### See Also

`SimBiology.export.Model` | `SimBiology.export.Model.getdose` |  
`SimBiology.export.SimulationOptions` | `SimBiology.export.ValueInfo` |  
`SimData` object | `export`

### Topics

“PK/PD Modeling and Simulation to Guide Dosing Strategy for Antibiotics”  
“Deploy a SimBiology Model”

# Species object

Options for compartment species

## Description

The SimBiology species object represents a *species*, which is a chemical or entity that participates in reactions, for example, DNA, ATP, Pi, creatine, G-Protein, or Mitogen-Activated Protein Kinase (MAPK). Species amounts can vary or remain constant during a simulation.

To add species that participate in reactions, add the reaction to the model. The process of adding the reaction to the model creates a compartment object (*unnamed*) and the necessary species objects.

Alternatively, create and add a species object to a compartment object, using the `addspecies` method at the command line.

When you use the SimBiology desktop to create a new model, it adds an empty compartment (*unnamed*), to which you can add species.

See “Property Summary” on page 2-498 for links to species property reference pages. Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the graphical user interface.

## Constructor Summary

`addspecies (model,  
compartment)`

Create species object and add to compartment object within model object

## Method Summary

Methods for species objects

copyobj (any object)	Copy SimBiology object and its children
delete (any object)	Delete SimBiology object
display (any object)	Display summary of SimBiology object
findUsages (species, parameter, compartment)	Find out how a species, parameter, or compartment is used in a model
get (any object)	Get object properties
rename (compartment, parameter, species, reaction)	Rename object and update expressions
set (any object)	Set object properties

## Property Summary

Properties for species objects

BoundaryCondition	Indicate species boundary condition
ConstantAmount	Specify variable or constant species amount
InitialAmount	Species initial amount
InitialAmountUnits	Species initial amount units
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

Compartment object, Configset object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object

**Introduced in R2006b**

## Unit object

Hold information about user-defined unit

### Description

The SimBiology unit object holds information about user-defined units. To create a unit, create the unit object and add the unit to the library using the `sbioaddtolibrary` function.

Use the unit object property `Composition` to specify the composition of your units. See “Property Summary” on page 2-501 for links to unit object property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change unit object properties in the SimBiology desktop. For more information, see “Libraries”.

### Constructor Summary

<code>sbiounit</code>	Create user-defined unit
-----------------------	--------------------------

### Method Summary

<code>delete (any object)</code>	Delete SimBiology object
<code>display (any object)</code>	Display summary of SimBiology object
<code>findUsages (unit, unitprefix)</code>	Find out how a unit or unit prefix is used
<code>get (any object)</code>	Get object properties
<code>set (any object)</code>	Set object properties



## Property Summary

Composition	Unit composition
Multiplier	Relationship between defined unit and base unit
Name	Specify name of object
Notes	HTML text describing SimBiology object
Offset	Unit composition modifier
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

AbstractKineticLaw object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object, Species object, UnitPrefix object

**Introduced in R2008a**

## UnitPrefix object

Hold information about user-defined unit prefix

### Description

The SimBiology unit prefix object holds information about user-defined unit prefixes. To create a unit prefix, create the unit prefix object and add the unit prefix to the library using the `sbiomaddtolibrary` function.

Use the unit prefix object property `Exponent`, to specify the exponent of your unit prefix. See “Property Summary” on page 2-503 for links to unit prefix object property reference pages.

Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change unit prefix object properties in the SimBiology desktop. For more information, see “Libraries”.

### Constructor Summary

<code>sbiomunitprefix</code>	Create user-defined unit prefix
------------------------------	---------------------------------

### Method Summary

<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>findUsages</code> (unit, unitprefix)	Find out how a unit or unit prefix is used
<code>get</code> (any object)	Get object properties
<code>set</code> (any object)	Set object properties

## Property Summary

Exponent	Exponent value of unit prefix
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

## See Also

AbstractKineticLaw object, KineticLaw object, Model object, Parameter object, Reaction object, Root object, Rule object, Species object, Unit object

**Introduced in R2008a**

# Variant object

Store alternate component values

## Description

The SimBiology variant object stores the names and values of model components and allows you to use the values stored in a variant object as the alternate value to be applied during a simulation. You can store values for species `InitialAmount`, parameter `Value`, and compartment `Capacity` in a variant object. Simulating using a variant does not alter the model component values. The values specified in the variant temporarily apply during simulation.

Using one or more variant objects associated with a model allows you to evaluate model behavior during simulation, with different values for the various model components without having to search and replace these values, or having to create additional models with these values. If you determine that the values in a variant object accurately define your model, you can permanently replace the values in your model with the values stored in the variant object, using the `commit` method.

To use a variant in a simulation you must add the variant object to the model object and set the `Active` property of the variant to true. Set the `Active` property to true if you always want the variant to be applied before simulating the model. You can also enter the variant object as an argument to `sbiosimulate`; this applies the variant only for the current simulation and supersedes any active variant objects on the model.

---

**Warning** The `Active` property of the `Variant` object will be removed in a future release. Explicitly specify a variant or an array of variants as an input argument when you simulate a model using `sbiosimulate`.

---

When there are multiple active variant objects on a model, if there are duplicate specifications for a property's value, the last occurrence for the property value in the array of variants, is used during simulation. You can find out which variant is applied last by looking at the indices of the variant objects stored on the model. Similarly, in the `Content` property, if there are duplicate specifications for a property's value, the last occurrence for the property in the `Content` property, is used during simulation.

Use the `addcontent` method to append contents to a variant object.

See “Property Summary” on page 2-505 for links to species property reference pages. Properties define the characteristics of an object. Use the `get` and `set` commands to list object properties and change their values at the command line. You can graphically change object properties in the graphical user interface.

## Constructor Summary

<code>sbiovariant</code>	Construct variant object
--------------------------	--------------------------

## Method Summary

Methods for variant objects

<code>addcontent</code> (variant)	Append content to variant object
<code>commit</code> (variant)	Commit variant contents to model
<code>copyobj</code> (any object)	Copy SimBiology object and its children
<code>delete</code> (any object)	Delete SimBiology object
<code>display</code> (any object)	Display summary of SimBiology object
<code>get</code> (any object)	Get object properties
<code>rmcontent</code> (variant)	Remove contents from variant object
<code>set</code> (any object)	Set object properties
<code>verify</code> (model, variant)	Validate and verify SimBiology model

## Property Summary

Properties for variant objects

Active	Indicate object in use during simulation
Content	Contents of variant object
Name	Specify name of object
Notes	HTML text describing SimBiology object
Parent	Indicate parent object
Tag	Specify label for SimBiology object
Type	Display SimBiology object type
UserData	Specify data to associate with object

### See Also

Compartment object, Configset object, Model object, Parameter object, Species object

`sbiosimulate`

**Introduced in R2008a**

# verify (model, variant)

Validate and verify SimBiology model

## Syntax

```
verify(modelObj)
```

```
verify(modelObj,csObj)
```

```
verify(modelObj,dvObj)
```

```
verify(modelObj,csObj,dvObj)
```

```
verify(modelObj,csObj,variantObj,doseObj)
```

## Description

`verify(modelObj)` performs checks on a Model `modelObj` to verify that you can simulate the model. This function generates stacked errors and warnings if it finds any problems. To see the entire list of errors and warnings, use `sbiolasterror` and `sbiolastwarning`. The function uses the active configuration set, any active doses and active variants for verification.

`verify(modelObj,csObj)` verifies a model `modelObj` using the specified configset object `csObj` and any active variants and active doses. Any other configsets are ignored. If you set `csObj` to empty `[]`, the function uses the active configset.

`verify(modelObj,dvObj)` verifies a model `modelObj` using doses or variants specified by `dvObj` and the active configset. `dvObj` can be one of the following:

- Variant object
- ScheduleDose object
- RepeatDose object
- Array of doses or variants

If you set `dvObj` to empty `[]`, the function uses the active configset, active variants, and active doses.

If you specify `dvObj` as variants, the function uses the specified variants and active doses. Any other variants are ignored.

If you specify `dvObj` as doses, the function uses the specified doses and active variants. Any other doses are ignored.

`verify(modelObj, csObj, dvObj)` verifies a model `modelObj` using a configset object `csObj` and doses or variants specified by `dvObj`.

If you set `csObj` to `[]`, then the function uses the active configset object.

If you set `dvObj` to `[]`, then the function uses no variants, but uses active doses.

If you specify `dvObj` as variants, the function uses the specified variants and active doses. Any other variants are ignored.

If you specify `dvObj` as doses, the function uses the specified doses and active variants. Any other doses are ignored.

`verify(modelObj, csObj, variantObj, doseObj)` verifies a model `modelObj` using a configset object `csObj`, variants (`variantObj`) and doses (`doseObj`). Any other configset, doses, and variants are ignored.

If you set `csObj` to `[]`, then the function uses the active configset object.

If you set `variantObj` to `[]`, then the function uses no variants.

If you set `doseObj` to `[]`, then the function uses no doses.

## Input Arguments

### **modelObj** — SimBiology model

SimBiology model object

SimBiology model, specified as a SimBiology model object.

### **csObj** — Configuration set object

configset object

Configuration set object, specified as a `configset` object that stores simulation-specific information.



**dvObj — Dose or variant object**

dose object or array of dose objects | variant object or array of variant objects

Dose or variant object, specified as a `ScheduleDose` object , `RepeatDose` object , an array of dose objects, `variant` object , or an array of variant objects.

- When `dvObj` is a dose object, `verify` uses the specified dose object as well as any active variant objects if available.
- When `dvObj` is a variant object, `verify` uses the specified variant object as well as any active dose objects if available.

**variantObj — Variant object**

variant object or array of variant objects

Variant object, specified as a `variant` object or an array of variant objects.

**doseObj — Dose object**

dose object or array of dose objects

Dose object, specified as a `ScheduleDose` object , `RepeatDose` object , or an array of dose objects. A dose object defines additions that are made to species amounts or parameter values.

## Examples

**Verify a SimBiology Model while Using a User-Defined Configset Object**

Load a sample SimBiology model.

```
sbioloadproject radiodecay.sbproj
```

Add a new configuration set using a different stop time of 15 seconds.

```
csObj = addconfigset(m1, 'newStopTimeConfigSet');
csObj.StopTime = 15;
```

Verify the model while using the configset object.

```
verify(m1, csObj);
```

After verification, check the latest errors and warnings if there is any.

```
sbiolasterror
```

```
ans =
```

```
    []
```

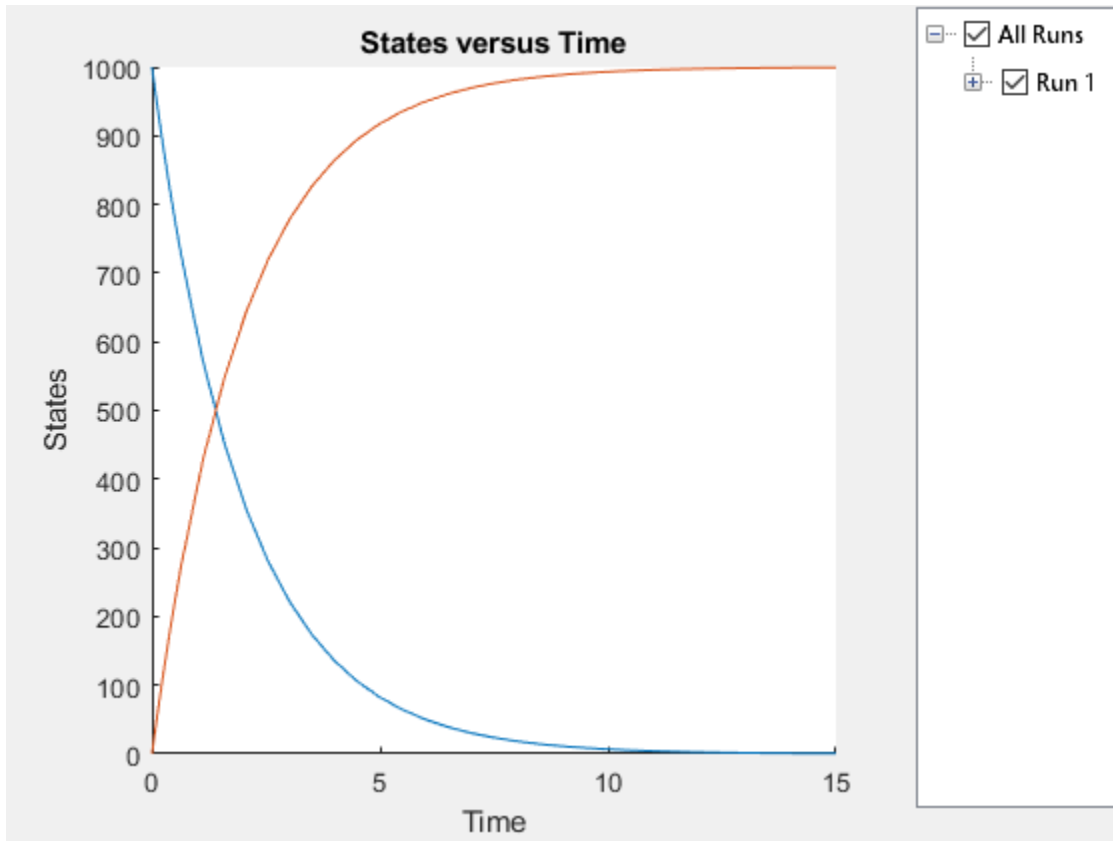
```
sbiolastwarning
```

```
ans =
```

```
    []
```

Simulate the model.

```
sim = sbiosimulate(m1,csObj);  
sbioplot(sim);
```



### Verify a SimBiology Model while Using Configset and Dose Objects

Load a sample SimBiology model.

```
sbioloadproject radiodecay.sbproj
```

Get the default configuration set from the model.

```
defaultConfigSet = getConfigset(m1, 'default');
```

Add a scheduled dose of 100 molecules at 2 seconds for species x.

```
dObj = adddose(m1, 'd1', 'schedule');
dObj.Amount = 100;
dObj.AmountUnits = 'molecule';
dObj.TimeUnits = 'second';
dObj.Time = 2;
dObj.TargetName = 'unnamed.x';
```

Verify the model while using the default configset object and added dose object.

```
verify(m1, defaultConfigSet, dObj);
```

After verification, check the latest errors and warnings if there is any.

```
sbiolasterror
```

```
ans =
```

```
    []
```

```
sbiolastwarning
```

```
ans = struct with fields:
```

```
    Type: 'Warning'
```

```
    MessageID: 'SimBiology:odebuilder:SENSITIVITY_OUTPUTS_WITH_REPEATED_ASSIGNMENT_VAR'
```

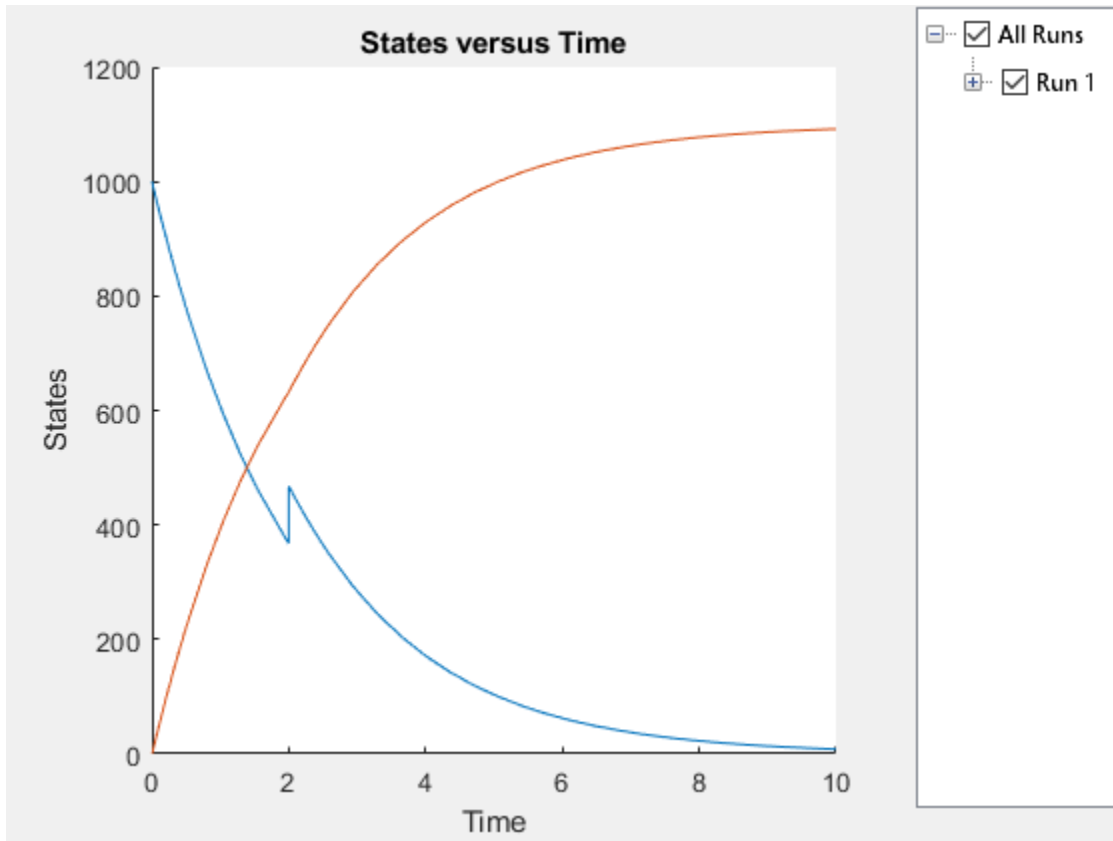
```
    Message: 'The SensitivityAnalysisOptions Outputs property contains objects that a'
```

Simulate the model using the same configset and dose objects.

```
sim = sbiosimulate(m1, defaultConfigSet, dObj);
```

Plot the result.

```
sbioplot(sim);
```



### Verify a SimBiology Model while Using Configset, Dose, and Variant Objects

Load a sample SimBiology model.

```
sbioloadproject radiodecay.sbproj
```

Add a new configuration set using a different stop time of 15 seconds.

```
csObj = m1.addconfigset('newStopTimeConfigSet');
csObj.StopTime = 15;
```

Add a scheduled dose of 100 molecules at 2 seconds for species x.

```
dObj = adddose(m1, 'd1', 'schedule');
dObj.Amount = 100;
dObj.AmountUnits = 'molecule';
dObj.TimeUnits = 'second';
dObj.Time = 2;
dObj.TargetName = 'unnamed.x';
```

Add a variant of species *x* using a different initial amount of 500 molecules.

```
vObj = addvariant(m1, 'v1');
addcontent(vObj, {'species', 'x', 'InitialAmount', 500});
```

Verify the model while using the configset, dose, and variant objects. Note that the order of arguments should be as described.

```
verify(m1, csObj, vObj, dObj);
```

After verification, check the latest errors and warnings if there is any.

```
sbiolasterror
```

```
ans =
```

```
    []
```

```
sbiolastwarning
```

```
ans = struct with fields:
```

```
    Type: 'Warning'
```

```
    MessageID: 'SimBiology:odebuilder:SENSITIVITY_OUTPUTS_WITH_REPEATED_ASSIGNMENT_VARS'
```

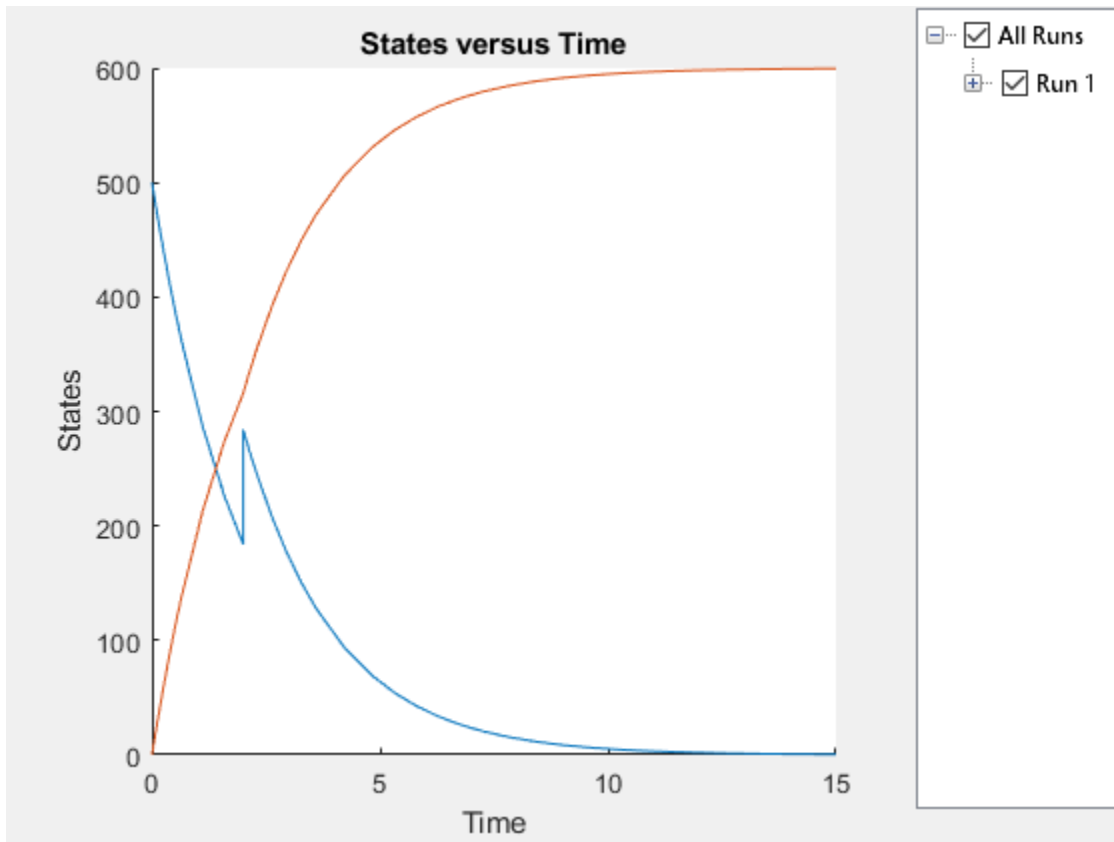
```
    Message: 'The SensitivityAnalysisOptions Outputs property contains objects that a
```

Simulate the model using the same configset, variant, and dose objects.

```
sim = sbiosimulate(m1, csObj, vObj, dObj);
```

Plot the result.

```
sbioplot(sim);
```



## See Also

`sbiolasterror`, `sbiolastwarning`

**Introduced in R2006a**

## verify (covmodel)

Check covariate model for errors

### Syntax

```
verify(CovModelObj)
```

### Description

`verify(CovModelObj)` verifies that the following are true about the Expression on page 3-85 property of *CovModelObj*, a `CovariateModel` object:

- The expressions are valid MATLAB code.
- Each expression is linear with a transformation.
- There is exactly one expression for each parameter.
- In each expression, a covariate is used in at most one term.
- In each expression, there is at most one random effect (`eta`)
- Fixed effect (`theta`) and random effect (`eta`) names are unique within and across expressions. That is, each covariate has its own fixed effect.

If the previous requirements are true, then `verify` returns nothing.

### See Also

`CovariateModel` on page 2-98 | `Expression` on page 3-85 | `PKModelDesign` object | `construct`

### Topics

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”  
“Specify a Covariate Model”

**Introduced in R2011b**



# Properties – Alphabetical List

---

## AbsoluteTolerance

Absolute error tolerance applied to state value during simulation

### Description

`AbsoluteTolerance` is a property of a `SolverOptions` object, which is a property of a `Configset` object. It is available for the ode solvers (`ode15s`, `ode23t`, `ode45`, and `sundials`).

SimBiology uses `AbsoluteTolerance` to determine the largest allowable absolute error at any step in a simulation. How the software uses `AbsoluteTolerance` to determine this error depends on whether the `AbsoluteToleranceScaling` property is enabled.

For details, see “Selecting Absolute Tolerance and Relative Tolerance for Simulation”.

### Characteristics

Applies to	Object: <code>SolverOptions</code>
Data type	<code>double</code>
Data values	Positive scalar. Default is <code>1e-6</code> .
Access	Read/write

### Examples

This example shows how to change `AbsoluteTolerance`.

- 1 Retrieve the `configset` object from the `modelObj`.

```
modelObj = sbiomodel('cell');  
configsetObj = getconfigset(modelObj)
```

- 2 Change the `AbsoluteTolerance` to `1e-8`.

```
set(configsetObj.SolverOptions, 'AbsoluteTolerance', 1.0e-8);  
get(configsetObj.SolverOptions, 'AbsoluteTolerance')
```

ans =

1.0000e-008

## See Also

AbsoluteToleranceScaling, AbsoluteToleranceStepSize, RelativeTolerance, MassUnits, AmountUnits, Configset object

## See Also

### Topics

“Model Simulation”

“Choosing a Simulation Solver”

“Ordinary Differential Equations” (MATLAB)

## AbsoluteToleranceScaling

Control scaling of absolute error tolerance during simulation

### Description

`AbsoluteToleranceScaling` is a property of a `SolverOptions` object, which is a property of a `Configset` object. It is available for the ode solvers (`ode15s`, `ode23t`, `ode45`, and `sundials`).

`AbsoluteToleranceScaling` controls how the software determines the largest allowable absolute error at any step in a simulation. For details, see “Selecting Absolute Tolerance and Relative Tolerance for Simulation”.

### Characteristics

Applies to	Object: <code>SolverOptions</code>
Data type	<code>logical</code>
Data values	<code>1</code> , <code>0</code> , <code>true</code> , or <code>false</code> . Default is <code>true</code> .
Access	Read/write

### See Also

`AbsoluteTolerance`, `AbsoluteToleranceStepSize`, `RelativeTolerance`

# AbsoluteToleranceStepSize

Initial guess for time step size for scaling of absolute error tolerance

## Description

AbsoluteToleranceStepSize is a property of a SolverOptions object, which is a property of a Configset object. It is available for the ode solvers (ode15s, ode23t, ode45, and sundials).

When the AbsoluteToleranceScaling property is enabled, you can set the AbsoluteToleranceStepSize property to specify the initial guess for time step size for scaling. For details, “Selecting Absolute Tolerance and Relative Tolerance for Simulation”.

---

**Tip** Use AbsoluteToleranceStepSize when a simulation is unsuccessful and generates numerically unstable solutions, and other corrective actions such as checking the model’s kinetics do not work. You might encounter unstable solutions if you have very stiff systems in which state values change rapidly at the beginning of a simulation. To solve this, iteratively decrease AbsoluteToleranceStepSize and simulate to find the optimal setting. As a starting point, try setting this property to  $\text{AbsoluteTolerance} * \text{StopTime} * 0.1$ .

---

## Characteristics

Applies to	Object: SolverOptions
Data type	double
Data values	Scalar in units specified by TimeUnits property. Default is [ ].
Access	Read/write

## See Also

AbsoluteTolerance, AbsoluteToleranceScaling, RelativeTolerance

## Active

Indicate object in use during simulation

## Description

The **Active** property indicates whether a simulation is using a SimBiology object. A SimBiology model is organized into a hierarchical group of objects. Use the **Active** property to include or exclude objects during a simulation.

- **Event, Reaction, or Rule** — When an event, a reaction, or rule object **Active** property is set to `false`, the simulation does not include the event, reaction, or rule. This is a convenient way to test a model with and without a reaction or rule.
- **Configuration set** — For the `configset` object, use the method `setactiveconfigset` to set the object **Active** property to `true`.

---

**Warning** This property of the `Configset` object will be removed in a future release. Explicitly specify a `configset` as an input argument when you simulate a model using `sbiosimulate`.

---

- **Variant** — Set the **Active** property to `true` if you always want the variant to be applied before simulating the model. You can also pass the variant object as an argument to `sbiosimulate`; this applies the variant only for the current simulation. For more information on using the **Active** property for variants, see `Variant` object.

---

**Warning** This property of the `Variant` object will be removed in a future release. Explicitly specify a variant or an array of variants as an input argument when you simulate a model using `sbiosimulate`.

---

- **Schedule dose** and **Repeat dose** — To use a dose object in a simulation, you must add the dose object to a model object and set the **Active** property of the dose object to `true`.

---

**Warning** This property of the `ScheduleDose` object and `RepeatDose` object will be removed in a future release. Explicitly specify a dose or an array of doses as an input argument when you simulate a model using `sbiosimulate`.

---

## Characteristics

Applies to	Objects: <code>configset</code> , <code>event</code> , <code>reaction</code> , <code>RepeatDose</code> , <code>rule</code> , <code>ScheduleDose</code> , <code>variant</code>
Data type	<code>boolean</code>
Data values	<code>true</code> or <code>false</code> . The default value for events, reactions, and rules is <code>true</code> . For the <code>configset</code> object, default is <code>true</code> . For added <code>configset</code> object, the default is <code>false</code> . For variants, the default is <code>false</code> .
Access	Read/write

## Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add a reaction object and verify that the `Active` property setting is `'true'` or `1`.

```
reactionObj = addreaction (modelObj, 'a + b -> c + d');
get (reactionObj, 'Active')
```

MATLAB returns:

```
ans =
```

```
1
```

- 3 Set the `Active` property to `'false'` and verify.

```
set (reactionObj, 'Active', false);
get (reactionObj, 'Active')
```

MATLAB returns:

ans =

0

## **See Also**

addconfigset, addreaction, addrule, Event object, Reaction object, RepeatDose object, Rule object, ScheduleDose object, Variant object,



# Amount

Amount of dose

## Description

Amount is a property of a `RepeatDose` or `ScheduleDose` object. It defines an increase in the amount of a SimBiology species that receives a dose.

A `RepeatDose` object defines a series of doses. Each dose is the same amount, as defined by the `Amount` property, and given at equally spaced times, as defined by the `Interval` property.

The number of injections in the series, excluding the initial injection, is defined by the `RepeatCount` property, and the `Rate` property defines how fast each dose is given.

For `RepeatDose` objects, you can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

A `ScheduleDose` object defines a series of doses. Each dose can have a different amount, as defined by an amount array in the `Amount` property, and given at specified times, as defined by a time array in the `Time` property. A rate array in the `Rate` property defines how fast each dose is given. At each time point in the time array, a dose is given with the corresponding amount and rate.

## Characteristics

Applies to	Object: <code>RepeatDose</code> , <code>ScheduleDose</code> .
Data type	double or character vector ( <code>RepeatDose</code> ) or double array ( <code>ScheduleDose</code> ).
Data values	Nonnegative value or name of a model-scoped parameter object. The default value is 0 ( <code>RepeatDose</code> ) or [] ( <code>ScheduleDose</code> ).
Access	Read/write.

## **See Also**

RepeatDose object | ScheduleDose object

## **Topics**

“Parameterized and Adaptive Doses”

# AmountUnits

Dose amount units

## Description

AmountUnits is a property of a RepeatDose or ScheduleDose object. This property defines units for the Amount property.

If the TargetName property defines a species, then AmountUnits for a dose must be a chemical amount (for example, milligram, mole, or molecule), not a concentration. To get a list of the defined units in the library, use the sbioshowunits function. To add a user-defined unit to the list, see sbioaddtolibrary.

## Characteristics

Applies to	Objects: RepeatDose, ScheduleDose
Data type	Character vector
Data values	Units from library with dimensions of amount. Default = "" (empty)
Access	Read/write

**Note** SimBiology uses units including empty units in association with DimensionalAnalysis and UnitConversion features.

- When DimensionalAnalysis and UnitConversion are both false, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.
- When DimensionalAnalysis is true and UnitConversion is false, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.
- When UnitConversion is set to true (which requires DimensionalAnalysis to be true), SimBiology performs a dimensional analysis and converts everything to

consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its unit to `dimensionless`.

---

## See Also

`ScheduleDose` object and `RepeatDose` object

## AmountUnits

Amount unit used internally during simulation when UnitConversion is on

### Description

This property defines the amount unit that SimBiology uses internally during model simulation when UnitConversion is on. You can set this to any character vector representing an amount unit such as molecule, mole, or mole with any valid prefix. It can also be a custom unit if it is consistent with amount as its dimension. The default is `<automatic>`, which means SimBiology automatically selects an amount unit for simulation. SimBiology examines the units on all of the states and selects an amount unit such that AbsoluteTolerance of the states in amount, or amount per volume is at least as stringent as the simulation absolute tolerance multiplied by the smallest amount unit. This stringency is relaxed appropriately for states that become large when AbsoluteToleranceScaling is on.

---

**Note** It is recommended that you use the default unit (`<automatic>`) or choose units for states such that the simulated values are neither too large (greater than  $10^6$ ) or too small (less than  $10^{-6}$ ).

However, for some edge cases, you may need to change AmountUnits. Suppose you have a model with a state that takes on values around  $10^{-12}$  moles for the entire simulation, and you need to use mole as its unit. Then it may be appropriate to set AmountUnits to picomole. In this case, the internal simulation values would be around 1, instead of around  $10^{-12}$  as in the default case. AbsoluteTolerance of the simulation is determined using this internal value. Thus by choosing picomole as the amount unit, you effectively reduce the size of AbsoluteTolerance. Changing the AmountUnits property is closely related to changing AbsoluteTolerance when considering the effects on simulation results.

Even when using the default unit, it may be still necessary to change AbsoluteTolerance. For details, see “Selecting Absolute Tolerance and Relative Tolerance for Simulation”.

If you need to recover the simulation behavior from releases prior to R2015b:

- Set the AmountUnit to mole. However, if the model has quantity units in molecule, set the unit to molecule instead.

- Set the MassUnits to kilogram.
- 

**Tip** If you have a custom function and UnitConversion is on (whether or not you are using the default unit <automatic>), follow the recommendation below.

- Non-dimensionalize the parameters that are passed to the function if they are not already dimensionless.

Suppose you have a custom function defined as  $y = f(t)$  where  $t$  is the time in hour and  $y$  is the concentration of a species in mole/liter. When you use this function in your model to define a repeated assignment rule for instance, define it as:  $s1 = f(\text{time}/t0) * s0$ , where  $\text{time}$  is the simulation time,  $t0$  is a parameter defined as 1.0 hour,  $s0$  is a parameter defined as 1.0 mole/liter, and  $s1$  is the concentration of a species in mole/liter. Note that  $\text{time}$  and  $s1$  do not have to be in the same units as  $t0$  and  $s0$ , but they must be dimensionally consistent. For example, the  $\text{time}$  and  $s1$  units can be set to minute and picomole/liter, respectively.

---

## Characteristics

Applies to	Object: Configset
Data type	Character vector
Data values	Character vector specifying any amount unit. The default is <automatic>.
Access	Read/write for properties of Configset

## See Also

Configset object, MassUnits

## Annotation

Store link to URL or file

---

**Note** The Annotation property will be removed in a future release. Use the Notes property instead.

---

## Description

The Annotation property stores the URL or file name linking to information about a model.

## Characteristics

Applies to	SimBiology objects: abstract kinetic law, configuration set, compartment, event, kinetic law, model, parameter, reaction, RepeatDose, rule, ScheduleDose, species, or unit
Data type	char string, URL
Data values	Character string with a directory path and filename or a URL
Access	Read/write

## Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Set the annotation for a model object.

```
set (modelObj, 'annotation', 'www.reactome.org')
```

- 3 Verify the assignment.

```
get (modelObj, 'annotation')
```

MATLAB returns:

```
ans =
```

```
www.reactome.org
```

## See Also

`addkineticlaw`, `addparameter`, `addreaction`, `addrule`, `addspecies`, `sbiomodel`, `sbioroot`, `sbiounit`, `sbiounitprefix`, `RepeatDose` object, `ScheduleDose` object



# BoundaryCondition

Indicate species boundary condition

## Description

The `BoundaryCondition` property indicates whether a species object has a boundary condition.

When the `BoundaryCondition` of a species is `false` (default), the species quantity is modified by reactions, rules, events, and doses. If the `BoundaryCondition` is `true`, the species quantity is modified by rules, events, and doses, but not by reactions.

Set the `BoundaryCondition` to `true` when you want the species to participate in a reaction, but do not want any reactions to modify its quantity.

All SimBiology species are state variables regardless of the `BoundaryCondition` or `ConstantAmount` on page 3-37 property.

## More Information

Consider the following two use cases of boundary conditions:

- Modeling receptor-ligand interactions that affect the rate of change of the receptor but not the ligand. For example, in response to hormone, steroid receptors such as the glucocorticoid receptor (GR) translocate from the cytoplasm (cyt) to the nucleus (nuc). The hsp90/ hsp70 chaperone complex directs this nuclear translocation [Pratt 2004 on page 3-22]. The natural ligand for GR is cortisol; the synthetic hormone dexamethasone (dex) is used in place of cortisol in experimental systems. In this system dexamethasone participates in the reaction but the quantity of dexamethasone in the cell is regulated using a rule. To simply model translocation of GR you could use the following reactions:

Formation of the chaperone-receptor complex,

```
Hsp90_complex + GR_cyt -> Hsp90_complex:GR_cyt
```

In response to the synthetic hormone dexamethasone (dex), GR moves from the cytoplasm to the nucleus.

```
Hsp90_complex:GR_cyt + dex -> Hsp90_complex + GR_nuc + dex
```

For dex,

```
BoundaryCondition = true; ConstantAmount = false
```

In this example dex is modeled as a boundary condition with a rule to regulate the rate of change of dex in the system. Here, the quantity of dex is not determined by the rate of the second reaction but by a rate rule such as

```
ddex/dt = 0.001
```

which is specified in the SimBiology software as

```
dex = 0.001
```

- Modeling the role of nucleotides (for example, GTP, ATP, cAMP) and cofactors (for example, Ca<sup>++</sup>, NAD<sup>+</sup>, coenzyme A). Consider the role of GTP in the activation of Ras by receptor tyrosine kinases.

```
Ras-GDP + GTP -> Ras-GTP + GDP
```

For GTP, BoundaryCondition = true; ConstantAmount = true

Model GTP and GDP with boundary conditions, thus making them *boundary species*. In addition, you can set the ConstantAmount on page 3-37 property of these species to true to indicate that their quantity does not vary during a simulation.

## Characteristics

Applies to	Object: species
Data type	boolean
Data values	true or false. The default value is false.
Access	Read/write

## Examples

## Simulate a Model with a Boundary Condition for a Species

This example illustrates how to use the `BoundaryCondition` property of a species so that the species amount is not modified by the reaction it participates in, but by a user-defined dose object.

Load a sample project.

```
sbioloadproject radiodecay.sbproj
```

A SimBiology model named `m1` is loaded to the MATLAB Workspace. The model is a simple radioactive decay model in which two species (`x` and `z`) are modified by the following reaction.

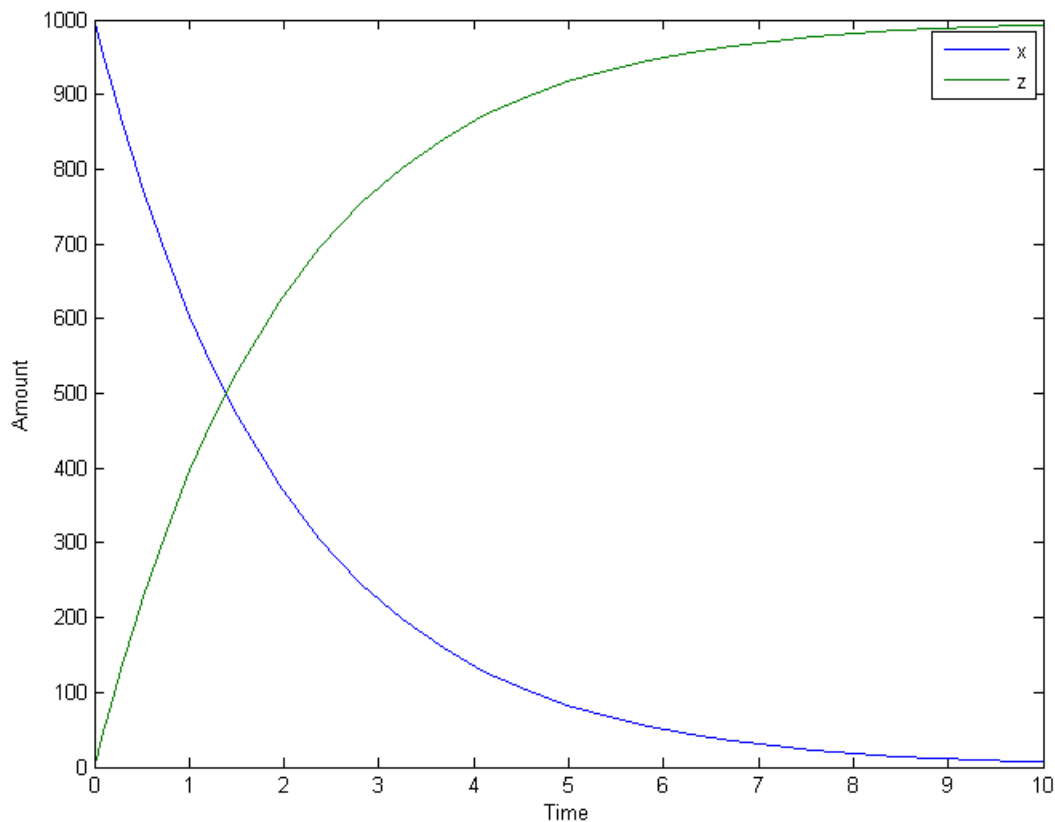
```
m1.Reactions
```

```
SimBiology Reaction Array
```

```
Index:    Reaction:  
1         x -> z
```

Simulate the model and view results before adding any boundary conditions.

```
[t,x,names] = sbiosimulate(m1);  
plot(t,x);  
legend(names)  
xlabel('Time');  
ylabel('Amount');
```



Add a RepeatDose object to the model and specify the species to be dosed, dose amount, dose schedule, and units.

```
d1 = adddose(m1, 'd1', 'repeat');  
set(d1, 'TargetName', 'z', 'Amount', 100.0, 'Interval', 1.0, 'RepeatCount', 8);  
set(d1, 'TimeUnits', 'second', 'AmountUnits', 'molecule');
```

Set the BoundaryCondition of species z to be true so that the species will be modified by the dose object d1, but not by the reaction.

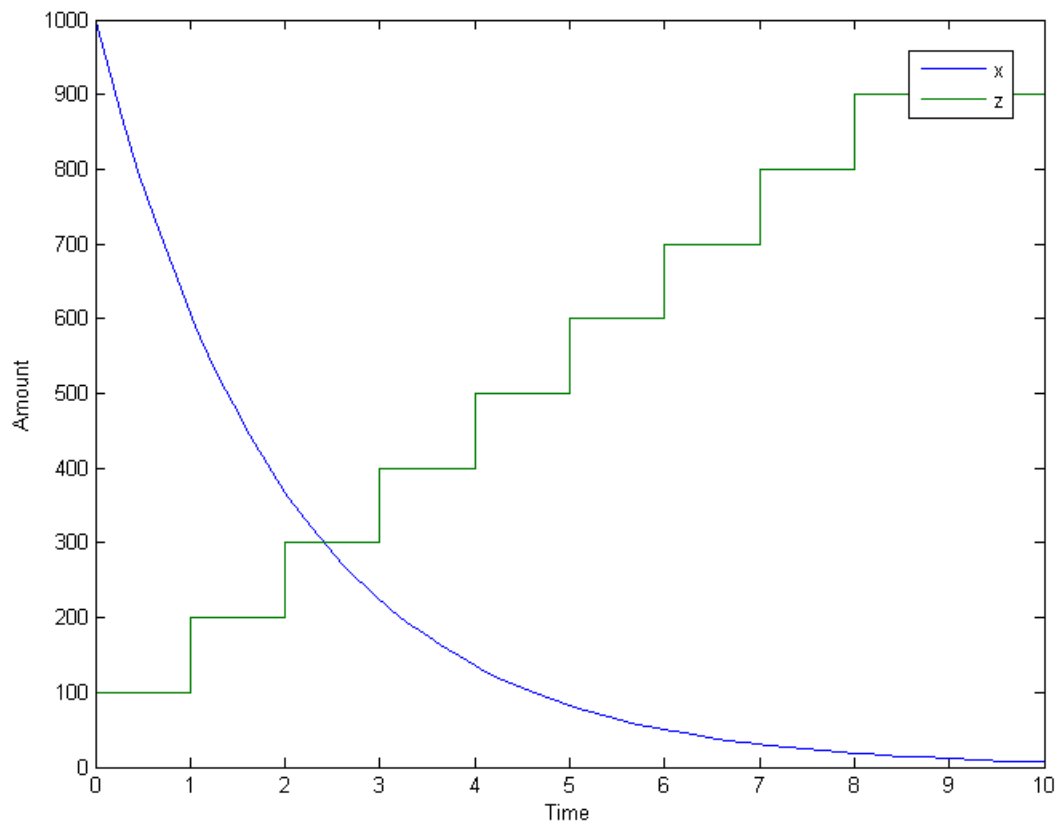
```
set(m1.species(2), 'BoundaryCondition', true);
```

Simulate the model by applying the dose object.

```
[t2,x2,names] = sbiosimulate(m1,d1);
```

Plot the results. Notice that the amount of species z is now modified by the repeated dose object, but not by the reaction.

```
[t2,x2,names] = sbiosimulate(m1,d1);  
plot(t2,x2);  
legend(names);  
xlabel('Time');  
ylabel('Amount');
```



## References

Pratt, W.B., Galigniana, M.D., Morishima, Y., Murphy, P.J. (2004), Role of molecular chaperones in steroid receptor action, *Essays Biochem*, 40:41-58.

## See Also

addrule, addspecies, ConstantAmount, InitialAmount

## BuiltInLibrary

Library of built-in components

### Description

**BuiltInLibrary** is a SimBiology root object property containing all built-in components namely, unit, unit-prefixes, and kinetic laws that are shipped with the SimBiology product. You cannot add, modify, or delete components in the built-in library. The **BuiltInLibrary** property is an object that contains the following properties:

- **Units** — contains all units that are shipped with the SimBiology product. You can specify units for compartment capacity, species amounts and parameter values, to do dimensional analysis and unit conversion during simulation. You can display the built-in units either by using the command `sbiowhos -builtin -unit`, or by accessing the root object.
- **UnitPrefixes** — contains all unit-prefixes that are shipped with the SimBiology product. You can specify unit—prefixes in combination with a valid unit for compartment capacity, species amounts and parameter values, to do dimensional analysis and unit conversion during simulation. You can display the built-in unit-prefixes either by using the command `sbiowhos -builtin -unitprefix`, or by accessing the root object.
- **KineticLaws** — contains all kinetic laws that are shipped with the SimBiology product. Use the command `sbiowhos -builtin -kineticlaw` to see the list of built-in kinetic laws. You can use built-in kinetic laws when you use the command `addkineticlaw` to create a kinetic law object for a reaction object. Refer to the kinetic law by name when you create the kinetic law object, for example, `kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');`

See “Kinetic Law Definition” on page 3-88 for a definition and more information.

### Characteristics

**BuiltInLibrary**



Applies to	Object: root
Data type	object
Data values	Unit, unit-prefix, and abstract kinetic law objects
Access	Read-only

Characteristics for `BuiltInLibrary` properties:

- **Units**

Applies to	<code>BuiltInLibrary</code> property
Data type	unit objects
Data values	units
Access	Read-only

- **UnitPrefixes**

Applies to	<code>BuiltInLibrary</code> property
Data type	unit prefix objects
Data values	unit prefixes
Access	Read-only

- **KineticLaws**

Applies to	<code>BuiltInLibrary</code> property
Data type	Abstract kinetic law object
Data values	kinetic laws
Access	Read-only

## Examples

### Example 1

This example uses the command `sbiowhos` to show the current list of built-in components.

```
sbiowhos -builtin -kineticlaw  
sbiowhos -builtin -unit  
sbiowhos -builtin -unitprefix
```

### Example 2

This example shows the current list of built-in components by accessing the root object.

```
rootObj = sbioroot;  
get(rootObj.BuiltinLibrary, 'KineticLaws')  
get(rootObj.BuiltinLibrary, 'Units')  
get(rootObj.BuiltinLibrary, 'UnitPrefixes')
```

### See Also

Functions — [sbioaddtolibrary](#), [sbioremovefromlibrary](#), [sbioroot](#), [sbiounit](#), [sbiounitprefix](#)

Properties — [UserDefinedLibrary](#)

# Capacity

Compartment capacity

## Description

The `Capacity` property indicates the size of the SimBiology compartment object. If the size of the compartment does not vary during simulation, set the property `ConstantCapacity` to `true`.

You can vary compartment capacity using rules or events.

---

**Note** Remember to set the `ConstantCapacity` property to `false` for varying capacity.

---

Events cannot result in the capacity having a negative value. Rules could result in the capacity having a negative value.

## Characteristics

Applies to	Object: compartment
Data type	<code>double</code>
Data values	Positive real number. The default value is 1.
Access	Read/write

## Examples

Add a compartment to a model and set the compartment capacity.

- 1 Create a model object named `my_model`.  

```
modelObj = sbiomodel ('comp_model');
```
- 2 Add the compartment object named `nucleus` with a capacity of 0.5.

```
compartmentObj = addcompartment(modelObj, 'nucleus', 0.5);
```

## **See Also**

`addcompartment`, `addspecies`, `CapacityUnits`, `ConstantCapacity`

# CapacityUnits

Compartment capacity units

## Description

The `CapacityUnits` property indicates the unit definition for the `Capacity` property of a compartment object. `CapacityUnits` can be any unit from the units library. To get a list of the defined units in the library, use the `sbioshowunits` function. If `CapacityUnits` changes from one unit definition to another, the `Capacity` does not automatically convert to the new units. The `sbioconvertunits` function does this conversion. To add a user-defined unit to the list, see `sbioaddtolibrary`.

## Characteristics

Applies to	Object: compartment
Data type	Character vector
Data values	Units from library with dimensions of length, area, or volume. Default = '' (empty).
Access	Read/write

---

**Note** SimBiology uses units including empty units in association with `DimensionalAnalysis` and `UnitConversion` features.

- When `DimensionalAnalysis` and `UnitConversion` are both `false`, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.
- When `DimensionalAnalysis` is `true` and `UnitConversion` is `false`, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.
- When `UnitConversion` is set to `true` (which requires `DimensionalAnalysis` to be `true`), SimBiology performs a dimensional analysis and converts everything to

consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its unit to `dimensionless`.

---

## Examples

- 1 Create a model object named `my_model`.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add a compartment object named `cytoplasm` with a capacity of `0.5`.

```
compObj = addcompartment (modelObj, 'cytoplasm', 0.5);
```

- 3 Set the `CapacityUnits` to `femtoliter`, and verify.

```
set (compObj, 'CapacityUnits', 'femtoliter');  
get (compObj, 'CapacityUnits')
```

MATLAB returns:

```
ans =
```

```
femtoliter
```

## See Also

`InitialAmount`, `sbioaddtolibrary`, `sbioconvertunits`, `sbioshowunits`

# Compartments

Array of compartments in model or compartment

## Description

Compartments shows you a read-only array of SimBiology compartment objects in the model object and the compartment object. In the model object, the `Compartments` property indicates all the compartments in a `Model` object as a flat list. In the compartment object, the `Compartments` property indicates other compartments that are referenced within the compartment. The two instances of `Compartments` are illustrated in “Examples” on page 3-31.

You can add a compartment object using the method `addcompartment`.

## Characteristics

Applies to	Objects: compartment, model
Data type	Array of compartment objects
Data values	Compartment object. Default is [] (empty).
Access	Read-only

## Examples

- 1 Create a model object named `modelObj`.

```
modelObj = sbiomodel('cell');
```

- 2 Add two compartments to the model object.

```
compartmentObj1 = addcompartment(modelObj, 'nucleus');
compartmentObj2 = addcompartment(modelObj, 'mitochondrion');
```

- 3 Add a compartment to one of the compartment objects.

```
compartmentObj3 = addcompartment(compartmentObj2, 'matrix');
```

- 4 Display the Compartments property in the model object.

```
get(modelObj, 'Compartments')
```

```
SimBiology Compartment Array
```

Index:	Name:	Capacity:	CapacityUnits:
1	nucleus	1	
2	mitochondrion	1	
3	matrix	1	

- 5 Display the Compartments property in the compartment object.

```
get(compartmentObj2, 'Compartments')
```

```
SimBiology Compartment - matrix
```

```
Compartment Components:  
Capacity: 1  
CapacityUnits:  
Compartments: 0  
ConstantCapacity: true  
Owner: mitochondrion  
Species: 0
```

## See Also

`addcompartment`, `addreaction`, `addspecies`, `Compartment` object



# CompileOptions

Dimensional analysis and unit conversion options

## Description

The SimBiology `CompileOptions` property is an object that defines the compile options available for simulation; you can specify whether dimensional analysis and unit conversion is necessary for simulation. Compile options are checked during compile time. The compile options object can be accessed through the `CompileOptions` property of the `configset` object. Retrieve `CompileOptions` object properties with the `get` function and configure the properties with the `set` function.

## Property Summary

<code>DefaultSpeciesDimension</code>	Dimension of species name in expression
<code>DimensionalAnalysis</code>	Perform dimensional analysis on model
<code>Type</code>	Display SimBiology object type
<code>UnitConversion</code>	Perform unit conversion

## Characteristics

Applies to	Object: <code>configset</code>
Data type	Object
Data values	Compile-time options
Access	Read-only

## Examples

- 1 Retrieve the `configset` object of `modelObj`.

```
modelObj = sbiomodel('cell');  
configsetObj = getconfigset(modelObj);
```

- 2 Retrieve the CompileOptions object (optionsObj) from the configsetObj.

```
optionsObj = get(configsetObj, 'CompileOptions');
```

Compile Settings:

```
UnitConversion:    false  
DimensionalAnalysis: true
```

## See Also

get, set

# Composition

Unit composition

## Description

The `Composition` property holds the composition of a unit object. The `Composition` property shows the combination of base and derived units that defines the unit. For example, molarity is the name of the unit and the composition is mole/liter. Base units are the set of units used to define all unit quantity equations. Derived units are defined using base units or mixtures of base and derived units.

Valid physical quantities for reaction rates are amount/time, mass/time, or concentration/time.

## Characteristics

Applies to	Object: Unit
Data type	Character vector
Data values	Valid combination of units and prefixes from the library. Default is '' (empty).
Access	Read/write

## Examples

This example shows you how to create a user-defined unit, add it to the user-defined library, and query the `Composition` property.

- 1 Create a unit for the rate constants of a second-order reaction.
 

```
unitObj = sbiounit('secondconstant', '1/molarity*second', 1);
```
- 2 Query the `Composition` property.
 

```
get(unitObj, 'Composition')
```

```
ans =  
1/molarity*second  
3 Change the Composition property.  
set(unitObj, 'Composition', 'liter/mole*second')  
ans =  
liter/mole*second  
4 Add the unit to the user-defined library.  
sbioaddtolibrary(unitObj);
```

## See Also

get, Multiplier, Offset, sbiounit, set

# ConstantAmount

Specify variable or constant species amount

## Description

The `ConstantAmount` property indicates whether the quantity of the species object can vary during the simulation. `ConstantAmount` can be either `true` or `false`. If `ConstantAmount` is `true`, the quantity of the species cannot vary during the simulation. By default, `ConstantAmount` is `false` and the quantity of the species can vary during the simulation. If `ConstantAmount` is `false`, the quantity of the species can be determined by reactions and rules.

The property `ConstantAmount` is for species objects; the property `ConstantValue` on page 3-41 is for parameter objects.

---

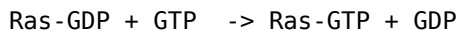
**Note** When you want the species to participate in a reaction, but do not want any reactions to modify its quantity, set its `BoundaryCondition` to `true`, and `ConstantAmount` to `false`.

---

## More Information

The following is an example of modeling species as constant amounts:

Modeling the role of nucleotides (GTP, ATP, cAMP) and cofactors (Ca<sup>++</sup>, NAD<sup>+</sup>, coenzyme A). Consider the role of GTP in the activation of Ras by receptor tyrosine kinases.



Model GTP and GDP with constant amount set to `true`. In addition, you can set the `BoundaryCondition` of these species to `true`, thus making them *boundary species*.

## Characteristics

Applies to	Object: species
Data type	boolean
Data values	true or false. The default value is false.
Access	Read/write

## Examples

- 1 Create a model object named `my_model`.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add a species object and verify that the `ConstantAmount` property setting is 'false' or 0.

```
speciesObj = addspecies (modelObj, 'glucose');  
get (speciesObj, 'ConstantAmount')
```

MATLAB returns:

```
ans =
```

```
0
```

- 3 Set the constant amount to 'true' and verify.

```
set (speciesObj, 'ConstantAmount', true);  
get (speciesObj, 'ConstantAmount')
```

MATLAB returns:

```
ans =
```

```
1
```

## See Also

`addspecies`, `BoundaryCondition`

# ConstantCapacity

Specify variable or constant compartment capacity

## Description

The `ConstantCapacity` property indicates whether the capacity of the compartment object can vary during the simulation. `ConstantCapacity` can be either `true` (1) or `false` (0). If `ConstantCapacity` is `true`, the quantity of the compartment cannot vary during the simulation. By default, `ConstantCapacity` is `true` and the quantity of the compartment cannot vary during the simulation. If `ConstantCapacity` is `false`, the quantity of the compartment can be determined by rules and events.

## Characteristics

Applies to	Object: compartment
Data type	boolean
Data values	true or false. The default value is true.
Access	Read/write

## Examples

Add a compartment to a model and check the `ConstantCapacity` property of the compartment.

- 1 Create a model object named `comp_model`.
 

```
modelObj = sbiomodel ('comp_model');
```
- 2 Add the compartment object named `nucleus` with a capacity of 0.5.
 

```
compartmentObj = addcompartment(modelObj, 'nucleus', 0.5);
```
- 3 Display the `ConstantCapacity` property.
 

```
get(compartmentObj, 'ConstantCapacity')
```

ans =

1

## **See Also**

addcompartment, ConstantAmount, ConstantValue



## ConstantValue

Specify variable or constant parameter value

### Description

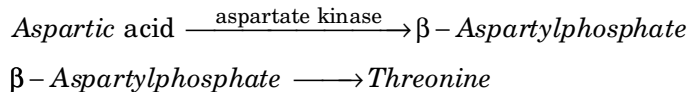
The `ConstantValue` property indicates whether the value of a parameter can change during a simulation. Enter either `true` (value is constant) or `false` (value can change).

You can allow the value of the parameter to change during a simulation by specifying a rule that changes the `Value` property of the parameter object.

The property `ConstantValue` is for parameter objects; the property `ConstantAmount` is for species objects.

### More Information

As an example, consider feedback inhibition of an enzyme such as aspartate kinase by threonine. Aspartate kinase has three isozymes that are independently inhibited by the products of downstream reactions (threonine, homoserine, and lysine). Although threonine is made through a series of reactions in the synthesis pathway, for illustration, the reactions are simplified as follows:



To model inhibition of aspartate kinase by threonine, you could use a rule like the algebraic rule below to vary the rate of the above reaction and simulate inhibition. In the rule, the rate constant for the above reaction is denoted by `k_aspartate_kinase` and the quantity of threonine is `threonine`.

```
k_aspartate_kinase -(1/threonine)
```

## Characteristics

Applies to	Object: parameter
Data type	boolean
Data values	true or false. The default value is 'true'.
Access	Read/write

## Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add a parameter object.

```
parameterObj = addparameter (modelObj, 'kf');
```

- 3 Change the ConstantValue property of the parameter object from default (true) to false and verify.

MATLAB returns 1 for true and 0 for false.

```
set (parameterObj, 'ConstantValue', false);  
get(parameterObj, 'ConstantValue')
```

MATLAB returns:

```
ans =
```

```
0
```

## See Also

addparameter

# Content

Contents of variant object

## Description

The `Content` property contains the data for the variant object. `Content` is a cell array with the structure `{'Type', 'Name', 'PropertyName', 'PropertyValue'}`. You can store values for species `InitialAmount`, parameter `Value`, and compartment `Capacity`, in a variant object.

For more information about variants, see `Variant` object.

## Characteristics

Applies to	Object: variant
Data type	cell array
Data values	Default value is [] (empty).
Access	Read/write

## Examples

- 1 Create a model containing three species in one compartment.

```
modelObj = sbiomodel('mymodel');
compObj = addcompartment(modelObj, 'comp1');
A = addspecies(compObj, 'A');
B = addspecies(compObj, 'B');
C = addspecies(compObj, 'C');
```

- 2 Add a variant object that varies the species' `InitialAmount` property.

```
variantObj = addvariant(modelObj, 'v1');
addcontent(variantObj, {'species','A', 'InitialAmount', 5}, ...
{'species', 'B', 'InitialAmount', 10});
% Display the variant
variantObj
```

SimBiology Variant - v1 (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	species	A	InitialAmount	5
2	species	B	InitialAmount	10

#### 3 Append data to the Content property.

```
addcontent(variantObj, {'species', 'C', 'InitialAmount', 15});
```

SimBiology Variant - v1 (inactive)

ContentIndex:	Type:	Name:	Property:	Value:
1	species	A	InitialAmount	5
2	species	B	InitialAmount	10
3	species	C	InitialAmount	15

#### 4 Remove a species from the Content property.

```
rmcontent(variantObj, 3);
```

#### 5 Replace the data in the Content property.

```
set(variantObj, 'Content', {'species', 'C', 'InitialAmount', 15});
```

## See Also

addcontent, rmcontent, sbiovariant

## CovariateLabels

Identify covariate columns in data set

### Description

CovariateLabels is a property of the PKData object. It specifies the column in DataSet on page 3-52 that contains the covariate data.

### Characteristics

Applies to	Object: PKData
Data type	Character vector or cell array of character vectors
Data values	Column headers from imported data set
Access	Read/write

### See Also

PKData object

## CovariateLabels (CovariateModel)

Labels for covariates in CovariateModel object

### Description

The CovariateLabels property is a cell array of character vectors specifying the labels for the covariates in the Expression on page 3-85 property of a CovariateModel object.

### Characteristics

Applies to	Object: CovariateModel on page 2-98
Data type	Cell array of character vectors
Data values	Labels for the covariates in the Expression on page 3-85 property
Access	Read only

### See Also

CovariateModel on page 2-98 | Expression on page 3-85

## Data

Store simulation data

## Description

The `Data` property contains the simulation data stored in the `SimData` object.

This property contains all data logged during a simulation, including species amounts, parameter values, and sensitivities. The property is an  $m \times n$  array, where  $m$  is the number of time steps in the simulation and  $n$  is the number of quantities logged. The rows of the array are labeled by the time points in the `Time` property, and the columns are labeled by the metadata in the `DataInfo` property.

## Characteristics

Applies to	Object: <code>SimData</code>
Data type	<code>double</code>
Data values	Default value is <code>[]</code> (empty).
Access	Read-only

## See Also

`DataInfo`, `ModelName`

## DataCount

Numbers of species, parameters, sensitivities

### Description

The `DataCount` property shows how many species, parameters, and sensitivities are logged in a `SimData` object. It is a MATLAB structure with the fields `Species`, `Parameter`, and `Sensitivity`. The information in this property is redundant with the `DataInfo` property and is there to give you a convenient means to access the information.

### Characteristics

Applies to	Object: <code>SimData</code>
Data type	<code>struct</code>
Data values	Default value for each field is 0.
Access	Read-only

### See Also

`StopTime`



## DataInfo

Metadata labels for simulation data

## Description

The DataInfo property contains the metadata that label the columns of the SimData object array. It is an  $n \times 1$  cell array of structures. The  $i$ th cell contains metadata labeling the  $i$ th column of the SimData object array.

The possible types of structures are as follows.

Type	Fields
Species	Type: species Name: Compartment: Units:
Parameter	Type: parameter Name: Reaction: <name of reaction that a parameter is scoped to, or '' if parameter is scoped to model> Units:
Sensitivity	Type: sensitivity Name: <for example: d[x]/d[y]_0> OutputType: <The type of the sensitivity output, either 'species' or 'parameter'> OutputName: <The name of the sensitivity output> OutputQualifier: <The compartment or reaction for the sensitivity output, for species or parameters, respectively> InputType: <The type of the sensitivity input, either 'species' or 'parameter'> InputName: <The name of the sensitivity input> InputQualifier: <The compartment or reaction for the sensitivity input, for species or parameters, respectively> Units:

## Characteristics

Applies to	Object: SimData
Data type	n x 1 cell array of structs
Data values	Default value is 0x1 cell array.
Access	Read-only

## See Also

StopTime

# DataNames

Show names in SimData object

## Description

The DataNames property holds the names that label the columns of the data matrix in the Data property. The property contains an nx1 character array. The software provides this information for your convenience.

## Characteristics

Applies to	Object: SimData
Data type	Character array
Data values	Default value is 0x1 cell array.
Access	Read-only

## See Also

StopTime

## DataSet

Dataset object containing imported data

### Description

DataSet is a property of the PKData object. It contains the imported data set. The PKData object constructor (PKData) assigns the specified data set to its DataSet property during construction.

### Characteristics

Applies to	Object: PKData
Data type	dataset object
Data values	Variable containing dataset object
Access	Read-only

### See Also

PKData object

# DefaultSpeciesDimension

Dimension of species name in expression

## Description

The `DefaultSpeciesDimension` property specifies how SimBiology interprets species names in expressions (namely reaction rate, rule, or event expressions). The valid property values are `substance` or `concentration`. If you specify `InitialAmountUnits`, SimBiology interprets species names appearing in expressions as concentration or substance amount according to the units specified, regardless of the value in `DefaultSpeciesDimension`. Thus, if `DefaultSpeciesDimension` is `concentration` and you specify species units as `molecule`, SimBiology interprets species names in expressions as `substance`. This interpretation applies even when `DimensionalAnalysis` is off.

You can find `DefaultSpeciesDimension` in the `CompileOptions` property.

When you set `DefaultSpeciesDimension` to `substance`, if you do not specify units, SimBiology interprets species names appearing in expressions as substance amounts, and does not scale by compartment capacity. To include a species concentration in an expression, divide by the appropriate compartment capacity in the expression. To specify compartment capacity in an expression enter the compartment name.

When you set `DefaultSpeciesDimension` to `concentration`, SimBiology interprets species names appearing in expressions as concentrations, and scales by compartment capacity in the expressions. To include a species amount in an expression, multiply by the species name by the appropriate compartment name in the expression.

For information on dimensional analysis for reaction rates, see “How Reaction Rates Are Evaluated”.

## Characteristics

Applies to	Object: <code>CompileOptions</code> (in configset object)
------------	---

Data type	Character vector
Data values	concentration or substance. Default value is concentration.
Access	Read/write

## See Also

CompileOptions, DimensionalAnalysis, get, getconfigset, sbiosimulate, set

# DependentVarLabel

Identify dependent variable column in data set

## Description

DependentVarLabel is a property of a PKData object. It specifies the column(s) in DataSet on page 3-52 that contain the dependent variable(s), for example, measured response(s). The column must contain numeric values, and cannot contain Inf or -Inf.

## Characteristics

Applies to	Object: PKData
Data type	Character vector or cell array of character vectors
Data values	Column header from an imported data set
Access	Read/write

## See Also

PKData object

## DependentVarUnits

Response units in PKData object

### Description

DependentVarUnits is a property of a PKData object. It specifies the units for the column(s) containing the dependent variable(s) (responses) in the imported data set. If unit conversion is on, plot results in the SimBiology desktop show the units specified in DependentVarUnits.

To get a list of units, use the `sbioshowunits` on page 1-286 function.

### Characteristics

Applies to	Object: PKData
Data type	Character vector or cell array of character vectors
Data values	Units from the units library. Default is an empty cell array.  <b>Tip</b> If there are no units associated with the dependent variable(s) in your data set, you can set this property to a cell array of empty character vectors, or simply an empty cell array.
Access	Read/write

### See Also

DependentVarLabel, PKData object



# DimensionalAnalysis

Perform dimensional analysis on model

## Description

The `DimensionalAnalysis` property specifies whether to perform dimensional analysis on the model before simulation. It is a property of the `CompileOptions` object. `CompileOptions` holds the model's compile time options and is the object property of the `configset` object. When `DimensionalAnalysis` is set to `true`, the SimBiology software checks whether the physical quantities of the units involved in reactions and rules, match and are applicable.

For example, consider a reaction  $a + b \rightarrow c$ . Using mass action kinetics, the reaction rate is defined as  $a*b*k$ , where  $k$  is the rate constant of the reaction. If you specify that initial amounts of  $a$  and  $b$  are 0.01M and 0.005M respectively, then units of  $k$  are  $1/(M*second)$ . If you specify  $k$  with another equivalent unit definition, for example,  $1/[(moles/liter)*second]$ , `DimensionalAnalysis` checks whether the physical quantities match. If the physical quantities do not match, you see an error and the model is not simulated.

Unit conversion requires dimensional analysis. If `DimensionalAnalysis` is off, and you turn `UnitConversion` on, then `DimensionalAnalysis` is turned on automatically. If `UnitConversion` is on and you turn off `DimensionalAnalysis`, then `UnitConversion` is turned off automatically.

If you have MATLAB function calls in your model, dimensional analysis ignores any expressions containing function calls and generates a warning.

Valid physical quantities for reaction rates are amount/time, mass/time, or concentration/time.

## Characteristics

Applies to	Object: <code>CompileOptions</code> (in <code>configset</code> object)
------------	--

Data type	boolean
Data values	true or false. Default value is true.
Access	Read/write

---

**Note** SimBiology allows exponentiation of any dimensionless quantity to any dimensionless power. For example, you can write the following expression if both `x` and `a` are dimensionless:  $(x + 3)^{(a + 0.5)}$

---

**Note** SimBiology uses units including empty units in association with `DimensionalAnalysis` and `UnitConversion` features.

- When `DimensionalAnalysis` and `UnitConversion` are both `false`, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.
  - When `DimensionalAnalysis` is `true` and `UnitConversion` is `false`, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.
  - When `UnitConversion` is set to `true` (which requires `DimensionalAnalysis` to be `true`), SimBiology performs a dimensional analysis and converts everything to consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its unit to `dimensionless`.
- 

**Tip** If you have a custom function and `UnitConversion` is on, follow the recommendation below.

- Non-dimensionalize the parameters that are passed to the function if they are not already dimensionless.

Suppose you have a custom function defined as  $y = f(t)$  where `t` is the time in hour and `y` is the concentration of a species in mole/liter. When you use this function in your model to define a repeated assignment rule for instance, define it as: `s1 = f(time/t0)*s0`, where `time` is the simulation time, `t0` is a parameter defined as 1.0 hour, `s0` is a parameter defined as 1.0 mole/liter, and `s1` is the concentration of a species in mole/liter. Note that `time` and `s1` do not have to be in the same units as `t0` and `s0`,

but they must be dimensionally consistent. For example, the `time` and `s1` units can be set to `minute` and `picomole/liter`, respectively.

---

## Examples

This example shows how to retrieve and set `DimensionalAnalysis` from the default `true` to `false` in the default configuration set in a model object.

- 1 Import a model.

```
modelObj = sbmlimport('oscillator')
```

```
SimBiology Model - Oscillator
```

```
Model Components:
```

```
Models:      0
Parameters:  0
Reactions:   42
Rules:       0
Species:     23
```

- 2 Retrieve the `configset` object of the model object.

```
configsetObj = getconfigset(modelObj)
```

```
Configuration Settings - default (active)
```

```
SolverType:      ode15s
StopTime:        10.000000
```

```
SolverOptions:
```

```
AbsoluteTolerance: 1.000000e-006
RelativeTolerance: 1.000000e-003
```

```
RuntimeOptions:
```

```
StatesToLog:      all
```

```
CompileOptions:
```

```
UnitConversion:   true
DimensionalAnalysis: true
```

- 3 Retrieve the `CompileOptions` object.

```
optionsObj = get(configsetObj, 'CompileOptions')
```

Compile Settings:

```
UnitConversion:    true
DimensionalAnalysis: true
```

- 4 Assign a value of false to DimensionalAnalysis.

```
set(optionsObj, 'DimensionalAnalysis', false)
```

## See Also

get, getconfigset, sbiosimulate, set

## Dosed

Dosed object name

## Description

Dosed is a property of the PKModelMap object. It specifies the name(s) of species object(s) that receive an input, such as a drug in a compartment or a ligand.

When dosing multiple compartments, a one-to-one relationship must exist between the number and order of elements in the Dosed property and the DosingType property.

## Characteristics

Applies to	Object: PKModelMap
Data type	Character vector or cell array of character vectors
Data values	Name of a species object or empty. Default is an empty cell array.
Access	Read/write

## See Also

DosingType, Estimated, Observed, PKModelMap object

## DoseLabel

Dose column in data set

### Description

DoseLabel is a property of the PKData object. DoseLabel specifies the column that contains the dosing information, in DataSet on page 3-52. The column must contain positive values, and cannot contain Inf or -Inf.

### Characteristics

Applies to	Object: PKData
Data type	Character vector or array of character vectors
Data values	Column headers from imported data set
Access	Read/write

### See Also

PKData object, sbionmimport, sbionmfiledef

## DoseUnits

Dose units in PKData object

### Description

The `DoseUnits` property indicates the units for dose values in the PKData object. Dose units must have dimensions of amount or mass. The length of `DoseUnits` must be the same as `DoseLabel`. For example, if the `DoseLabel` property defines two columns containing dosing information, `DoseUnits` must also define units for both columns. If unit conversion is on, dose and rate units must be consistent with each other (that is in terms of amount or mass) and must be consistent with the species object that is being dosed.

To get a list of units, use the `sbioshowunits` on page 1-286 function.

### Characteristics

Applies to	Object: PKData
Data type	Character vector or cell array of character vectors
Data values	Units from units library. Default is ' ' (empty).
Access	Read/write

### See Also

`DoseLabel`, PKData object

## DosingType

Drug dosing type in compartment

### Description

DosingType is a property of the PKCompartment and PKModelMap objects. It specifies the type of dosing of a drug in a compartment. You can only dose one compartment in the model at any given time. For a description of the types of dosing supported, the model components created for each type of dosing, and the parameters to estimate, see “Dosing Types”.

### Characteristics

Applies to	Objects: PKCompartment, PKModelMap
Data type	Character vector or cell array of character vectors
Data values	'', 'Bolus', 'Infusion', 'ZeroOrder', 'FirstOrder'
Access	Read/write

### See Also

EliminationType, PKCompartment object, PKModelMap object



# DurationParameterName

Parameter specifying length of time to administer a dose

## Description

DurationParameterName is a property of a RepeatDose or ScheduleDose object.

Specify the name of a parameter object that is scoped to a model. This parameter defines the length of time it takes to administer a dose.

you can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

---

**Note** If you set the DurationParameterName property of a dose, you must also specify the Amount property of the dose, and set the Rate property to 0. This is because the rate is calculated from the amount and duration.

---

## Characteristics

Applies to	Objects: RepeatDose, ScheduleDose.
Data type	Character vector.
Data values	Name of a model-scoped parameter object. The default value is an empty character vector ' '.
Access	Read/write.

## Examples

### Estimate Time Lag and Duration of a Dose

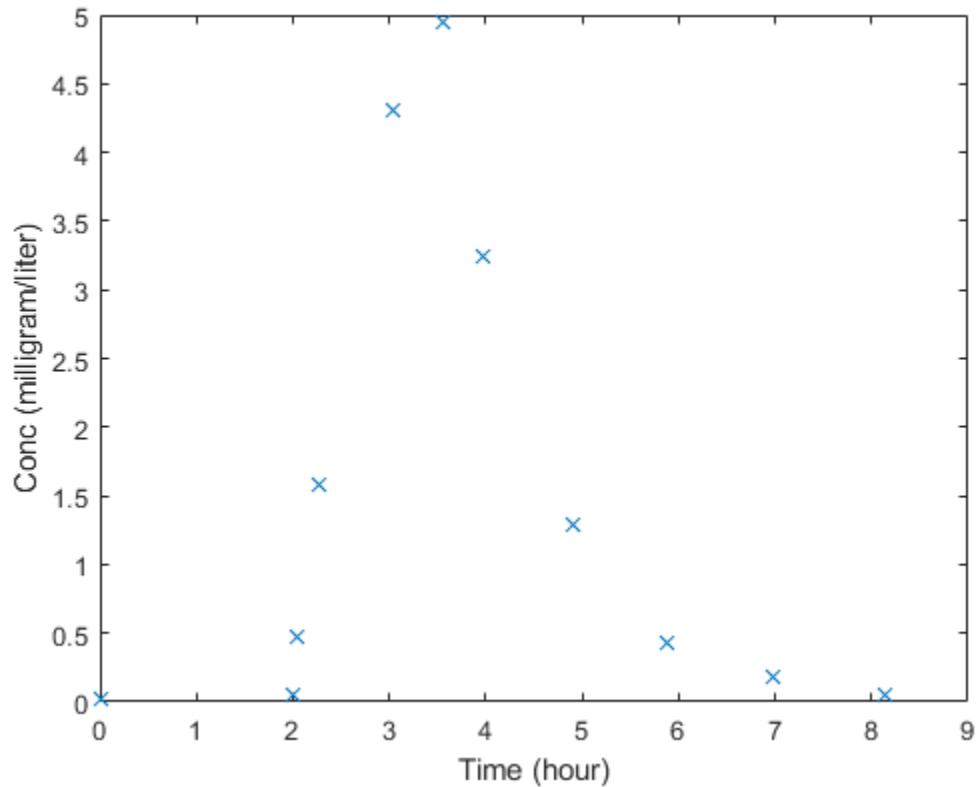
This example shows how to estimate the time lag before a bolus dose was administered and the duration of the dose using a one-compartment model.

Load a sample data set.

```
load lagDurationData.mat
```

Plot the data.

```
plot(data.Time,data.Conc,'x')  
xlabel('Time (hour)')  
ylabel('Conc (milligram/liter)')
```



Convert to groupedData.

```
gData = groupedData(data);  
gData.Properties.VariableUnits = {'hour','milligram/liter'};
```

Create a one-compartment model.

```
pkmd          = PKModelDesign;
pkc1          = addCompartment(pkmd, 'Central');
pkc1.DosingType = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
model        = construct(pkmd);
configset    = getConfigset(model);
configset.CompileOptions.UnitConversion = true;
```

Add two parameters that represent the time lag and duration of a dose. The lag parameter specifies the time lag before the dose is administered. The duration parameter specifies the length of time it takes to administer a dose.

```
lagP = addparameter(model, 'lagP');
lagP.ValueUnits = 'hour';
durP = addparameter(model, 'durP');
durP.ValueUnits = 'hour';
```

Create a dose object. Set the LagParameterName and DurationParameterName properties of the dose to the names of the lag and duration parameters, respectively.

```
dose          = sbiodose('dose');
dose.TargetName = 'Drug_Central';
dose.StartTime = 0;
dose.Amount    = 10;
dose.AmountUnits = 'milligram';
dose.TimeUnits = 'hour';
dose.LagParameterName = 'lagP';
dose.DurationParameterName = 'durP';
```

Map the model species to the corresponding data.

```
responseMap = {'Drug_Central = Conc'};
```

Specify the lag and duration parameters as parameters to estimate. Log-transform the parameters. Initialize them to 2 and set the upper bound and lower bound.

```
paramsToEstimate = {'log(lagP)', 'log(durP)'};
estimatedParams  = estimatedInfo(paramsToEstimate, 'InitialValue', 2, 'Bounds', [1 5]);
```

Perform parameter estimation.

```
fitResults = sbiofit(model, gData, responseMap, estimatedParams, dose, 'fminsearch')
```

```
fitResults =
  OptimResults with properties:

      ExitFlag: 1
      Output: [1x1 struct]
      GroupName: One group
      Beta: [2x4 table]
  ParameterEstimates: [2x4 table]
      J: [11x2 double]
      COVB: [2x2 double]
  CovarianceMatrix: [2x2 double]
      R: [11x1 double]
      MSE: 0.0024
      SSE: 0.0213
      Weights: []
  LogLikelihood: 18.7511
      AIC: -33.5023
      BIC: -32.7065
      DFE: 9
      DependentFiles: {1x2 cell}
  EstimatedParameterNames: {'lagP' 'durP'}
      ErrorModelInfo: [1x3 table]
  EstimationFunction: 'fminsearch'
```

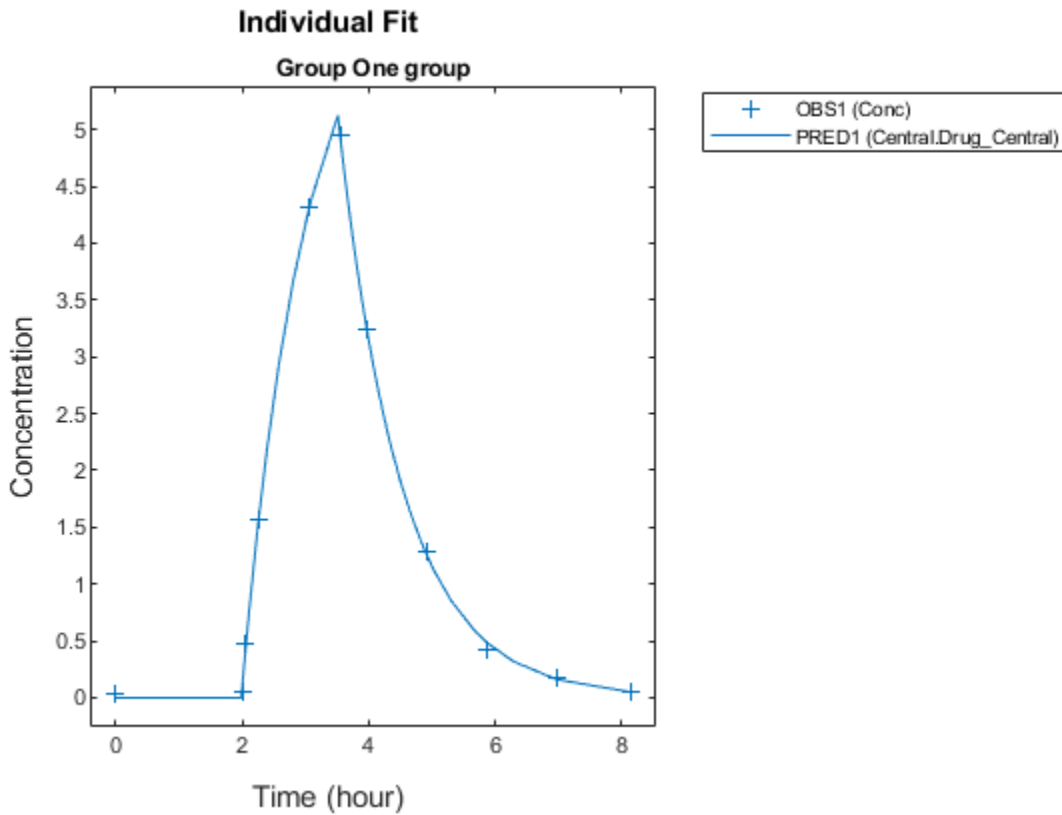
Display the result.

```
fitResults.ParameterEstimates
```

```
ans=2x4 table
```

Name	Estimate	StandardError	Bounds	
'lagP'	1.986	0.0051568	1	5
'durP'	1.527	0.012956	1	5

```
plot(fitResults)
```



## See Also

RepeatDose object | ScheduleDose object

## Topics

"Parameterized and Adaptive Doses"

## EliminationType

Drug elimination type from compartment

### Description

EliminationType is a property of the PKCompartment object. It specifies the type of elimination of a drug from a compartment. For a description of the types of elimination supported, the model components created for each type of elimination, and the parameters to estimate, see “Elimination Types”.

### Characteristics

Applies to	Object: PKCompartment
Data type	Character vector
Data values	'Linear', 'Linear-Clearance', 'Enzymatic', and ''
Access	Read/write

### See Also

addCompartment, DosingType, PKCompartment object

# ErrorTolerance

Specify explicit or implicit tau error tolerance

## Description

The `ErrorTolerance` property specifies the error tolerance for the explicit tau and implicit tau stochastic solvers. It is a property of the `SolverOptions` object. `SolverOptions` is a property of the `configset` object. The explicit and implicit tau solvers automatically chooses a time interval (`tau`) such that the relative change in the propensity function for each reaction is less than the user-specified error tolerance.

A propensity function describes the probability that the reaction will occur in the next smallest time interval, given the conditions and constraints.

If the error tolerance is too large, there may not be a solution to the problem and that could lead to an error. If the error tolerance is small, the solver will take more steps than when the error tolerance is large leading to longer simulation times. The error tolerance should be adjusted depending upon the problem, but a good value for the error tolerance is between 1 % to 5 %.

## Characteristics

Applies to	Object: <code>SolverOptions</code>
Data type	<code>double</code>
Data values	>0, <1. The default is <code>3e-2</code> .
Access	Read/write

## Examples

This example shows how to change `ErrorTolerance` settings.

- 1 Retrieve the `configset` object from the `modelObj` and change the `SolverType` to `expltau`.

```
modelObj = sbiomodel('cell');  
configsetObj = getConfigset(modelObj);  
set(configsetObj, 'SolverType', 'expltau')
```

- 2 Change the ErrorTolerance to 1e-8.

```
set(configsetObj.SolverOptions, 'ErrorTolerance', 5.0e-2);  
get(configsetObj.SolverOptions, 'ErrorTolerance')
```

```
ans =
```

```
5.000000e-002
```

## See Also

LogDecimation, RandomState



---

## Estimated

Names of parameters to estimate

## Description

Estimated is a property of the PKModelMap object. It specifies the name(s) of the object(s) to estimate. Specify the name(s) of species, compartment, or parameter object(s) that are scoped to a model.

---

**Note** If you specify a species object, you are estimating the InitialAmount property of the species object.

---

## Characteristics

Applies to	Object: PKModelMap
Data type	Character vector or cell array of character vectors
Data values	Name of a species, compartment, or parameter object or empty. Default is an empty cell array.
Access	Read/write

## See Also

Dosed, InitialAmount, Observed, PKModelMap object

## EventFcns

Event expression

### Description

Property of the event object that defines what occurs when the event is triggered. Specify a cell array of character vectors.

EventFcns can be any MATLAB assignment or expression that defines what is executed when the event is triggered. All EventFcn expressions are assignments of the form '*objectname* = *expression*', where *objectname* is the name of a valid SimBiology object.

For more information about how SimBiology handles events, see “How Events Are Evaluated”. For examples of event functions, see “Specifying Event Functions”.

### Characteristics

Applies to	Object: event
Data type	Cell array of character vectors
Data values	Expressions for EventFcn (default is an empty character vector '')
Access	Read/write

---

**Tip** If UnitConversion is on and your model has any event, follow the recommendation below.

Non-dimensionalize any parameters used in the event trigger if they are not already dimensionless. For example, suppose you have a trigger  $x > 1$ , where  $x$  is the species concentration in mole/liter. Non-dimensionalize  $x$  by scaling (dividing) it with a constant such as  $x/x_0 > 1$ , where  $x_0$  is a parameter defined as 1.0 mole/liter. Note that  $x$  does not have to have the same unit as the constant  $x_0$ , but must be dimensionally consistent with it. For example, the unit of  $x$  can be picomole/liter instead of mole/liter.

---

## Examples

- 1 Create a model object, and then add an event object.

```
modelObj = sbmlimport('oscillator');  
eventObj = addevent(modelObj, 'time>= 5', 'OpC = 200');
```

- 2 Set the EventFcns property of the event object.

```
set(eventObj, 'EventFcns', {'pA = 0pA', 'mA = pol'});
```

- 3 Get the EventFcns property.

```
get(eventObj, 'EventFcns')
```

## See Also

Event, Trigger

## EventMode

Determine how events that change dose parameters affect in-progress dosing

### Description

EventMode is a property of RepeatDose and ScheduleDose objects. This property determines whether to continue the ongoing dose, that is, a dose with a nondefault infusion rate or dose duration, to completion when an event changes a parameter that is referenced by a dose property. Dose properties that you can parameterize are: Amount, Rate, Interval, StartTime, RepeatCount, LagParameterName, and DurationParameterName.

If EventMode is set to 'continue', the ongoing dose continues to completion when the event changes dose parameters, and updated parameters affect only subsequent doses. If EventMode is set to 'stop', the ongoing dose stops immediately when dose parameters change, and subsequent doses use the updated parameters.

To decide whether a parameter has been changed, SimBiology compares the old value of a parameter to the new value. For instance, the following event does not change the doseStartTime parameter value: `addevent(model, 'time >= 5', 'doseStartTime = doseStartTime * 1')`.

Any change in dose parameters affects the schedule of doses generated. If the simulation reaches a time point for a scheduled dose, the dose is applied. If an event changes dose parameters, SimBiology updates the dose schedule, ignores any doses scheduled before the current simulation time, and applies only the subsequent doses. Suppose that you have parameterized the StartTime property of a dose. Updating the parameter with an event causes to regenerate the dose schedule. If there are any previously-scheduled doses before the current simulation time, they are ignored.

By default, SimBiology uses the following MATLAB expression to generate a list of dose times (dose schedule) whenever an event changes any dose parameter, using the updated parameter values:

```
scheduledDoseTimes = StartTime + (0:RepeatCount) * Interval + Lag
,
```

where *StartTime*, *RepeatCount*, and *Interval* are properties of the dose object. *Lag* is the time lag parameter for a dose, referenced by the *LagParameterName* property.

## Characteristics

Applies to	Objects: RepeatDose, ScheduleDose
Data type	Character vector
Data values	'stop' (default) or 'continue'
Access	Read/write

## Examples

### Change Dose Behavior In Response to Changes in Model Parameters

Create a simple model with linear elimination, an amount parameter, and a rate parameter.

```

model = sbiomodel('simple model');
compartment = addcompartment(model, 'Central', 1);
compartment.CapacityUnits = 'liter';
species = addspecies(model, 'drug');
species.InitialAmountUnits = 'milligram';

% Elimination rate
elimParam = addparameter(model, 'kel', 0.1);
elimParam.ValueUnits = '1/hour';

% Elimination reaction
reaction = addreaction(model, 'drug -> null');
reaction.ReactionRate = 'kel*drug';

% Add amount and rate parameters
amountParam = addparameter(model, 'A', 50);
amountParam.ConstantValue = false;
amountParam.ValueUnits = 'milligram'

```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	A	50	milligram

```
rateParam          = addparameter(model, 'R', 10);  
rateParam.ValueUnits = 'milligram/hour'
```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	R	10	milligram/hour

Create a dose with its **Amount** and **Rate** properties set to the amount and rate parameters 'A' and 'R', respectively.

```
dose                = adddose(model, 'adaptive dose', 'repeat');  
dose.Amount         = 'A';  
dose.Rate           = 'R';
```

Set other dose properties.

```
dose.TargetName     = 'drug';  
dose.StartTime      = 0;  
dose.TimeUnits      = 'hour';  
dose.Interval       = 24;  
dose.RepeatCount    = 7;
```

Prepare the configuration set to simulate the model for 7 days.

```
configset           = getconfigset(model);  
configset.StopTime  = 7*24;  
configset.TimeUnits = 'hour';
```

Add an event to reset the dose amount to 10 at time  $\geq 26$ .

```
event = addevent(model, 'time  $\geq 26$ ', 'A = 10');
```

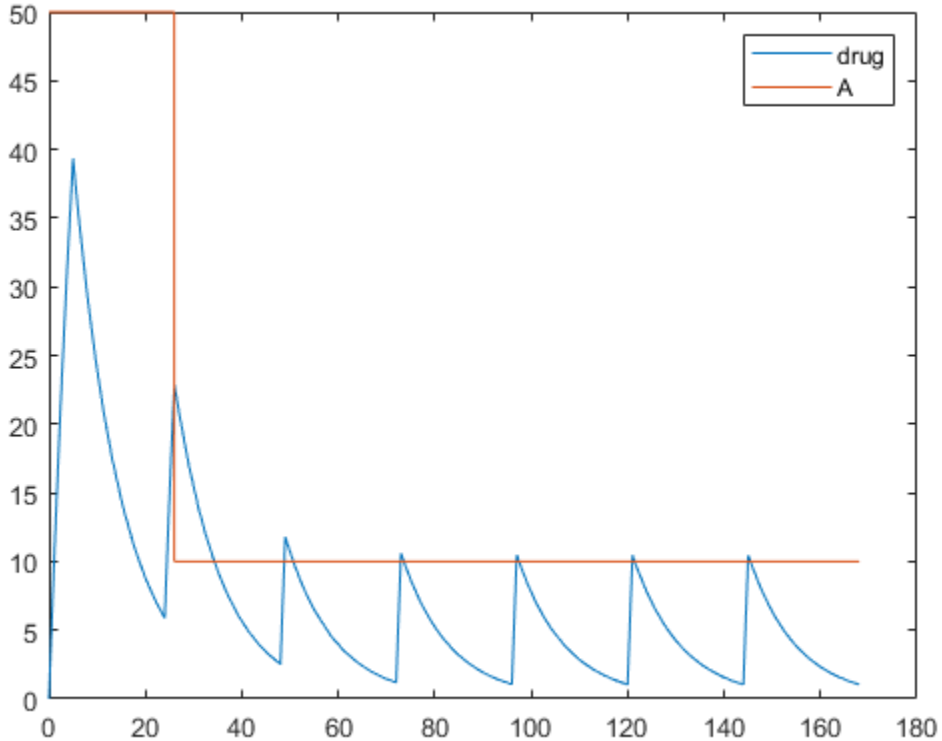
Set the **EventMode** property to 'stop'. This setting causes any ongoing dose event to stop at 26 hours.

```
dose.EventMode = 'stop';
```

Simulate the model. The second dose event stops at 26 hours, and the subsequent dose events continue with the new dose amount of 10.

```
[time, drugAndAmount] = sbiosimulate(model, dose);  
figure
```

```
plot(time, drugAndAmount);
legend('drug', 'A');
```

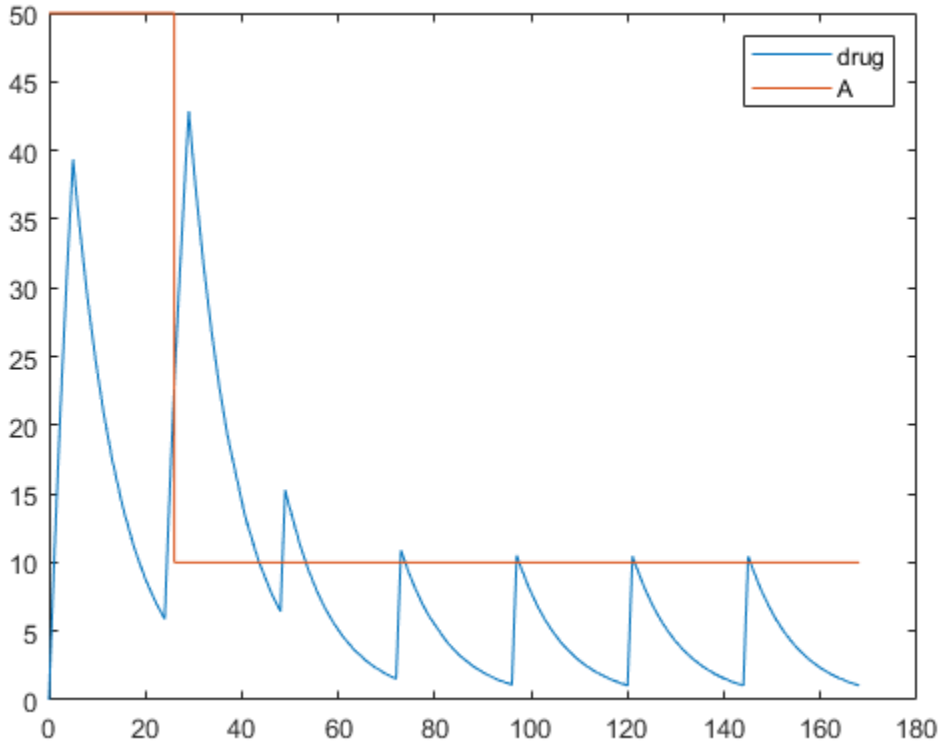


Alternatively, you can allow the ongoing dose event to finish before applying the new dose amount by setting `EventMode` to `'continue'`.

```
dose.EventMode = 'continue';
```

Simulate the model. In this case, the second dose event continues to 26 hours.

```
[time, drugAndAmount] = sbiosimulate(model, dose);
figure
plot(time, drugAndAmount);
legend('drug', 'A');
```



## See Also

RepeatDose object | ScheduleDose object

## Topics

“Parameterized and Adaptive Doses”



## Events

Contain all event objects

## Description

Property to indicate events in a model object. Read-only array of `Event` objects.

An event defines an action when a defined condition is met. For example, the quantity of a species may double when the quantity of species B is 100. An event is triggered when the conditions specified in the event are met by the model. For more information, see “Events” and “Events”.

Add an event to a `Model` object with the method `addevent (model)` method and remove an event with the `delete (any object)` method. See `Event` for more information.

You can view event object properties with the `get` command and modify the properties with the `set` command.

## Characteristics

Applies to	Object: model
Data type	Array of event objects
Data values	Event object. The default is [] (empty).
Access	Read-only

## Examples

- 1 Create a model object, and then add an event object.

```
modelObj = sbmlimport('oscillator')
eventObj = addevent(modelObj, 'time>= 5', 'OpC = 200');
```

- 2 Get a list of properties for an event object.

```
get(modelObj.Events(1));
```

Or

```
get(eventObj)
```

MATLAB displays a list of event properties.

```
    Active: 1
  Annotation: ''
   EventFcns: {'OpC = 200'}
      Name: ''
      Notes: ''
   Parent: [1x1 SimBiology.Model]
      Tag: ''
   Trigger: 'time >= 5'
  TriggerDelay: 0
TriggerDelayUnits: 'second'
      Type: 'event'
   UserData: []
```

## See Also

EventFcns, Event, Model, Trigger

# Exponent

Exponent value of unit prefix

## Description

*Exponent* shows the value of  $10^{\text{Exponent}}$  that defines the numerical value of the unit prefix *Name*. You can use the unit prefix in conjunction with any built-in or user-defined units. For example, for the unit mole, specify as `picomole` to use the Exponent, -12.

## Characteristics

Applies to	Object: Unit prefix
Data type	double
Data values	Real number. Default is 0.
Access	Read/write

## Examples

This example shows you how to create a user-defined unit prefix, add it to the user-defined library, and query the Exponent property.

- 1 Create a unit prefix.

```
unitprefixObj1 = sbiounitprefix('peta', 15);
```

- 2 Add the unit prefix to the user-defined library.

```
sbioaddtolibrary(unitprefixObj1);
```

- 3 Query the Exponent property.

```
get(unitprefixObj1, 'Exponent')
```

```
ans =
```

```
15
```

## **See Also**

get, sbioaddtolibrary, sbiounitprefix, set, UnitPrefix object

## Expression (CovariateModel)

Define relationship between parameters and covariates

### Description

The `Expression` property is a character vector or cell array of character vectors, where each character vector represents the relationship between a parameter and one or more covariates. The `Expression` property denotes fixed effects with the prefix `theta`, and random effects with the prefix `eta`.

Each expression must be in the form:  
`parameterName = relationship`

This example of an expression defines the relationship between a parameter (`volume`) and a covariate (`weight`), with fixed effects, but no random effects:  
`CovModelObj.Expression = {'volume = theta1 + theta2*weight'};`

This table illustrates expression formats for some common parameter-covariate relationships.

Parameter-Covariate Relationship	Expression Format
Linear with random effect	$Cl = \theta_1 + \theta_2 * WEIGHT + \eta_1$
Exponential without random effect	$Cl = \exp(\theta_{Cl} + \theta_{Cl\_WT} * WEIGHT)$
Exponential, <code>WEIGHT</code> centered by mean, has random effect	$Cl = \exp(\theta_1 + \theta_2 * (WEIGHT - \text{mean}(WEIGHT)) + \eta_1)$
Exponential, $\log(WEIGHT)$ , which is equivalent to power model	$Cl = \exp(\theta_1 + \theta_2 * \log(WEIGHT) + \eta_1)$
Exponential, dependent on <code>WEIGHT</code> and <code>AGE</code> , has random effect	$Cl = \exp(\theta_1 + \theta_2 * WEIGHT + \theta_3 * AGE + \eta_1)$

Parameter-Covariate Relationship	Expression Format
Inverse of probit, dependent on WEIGHT and AGE, has random effect	$Cl = \text{probitinv}(\text{theta1} + \text{theta2} * \text{WEIGHT} + \text{theta3} * \text{AGE} + \text{eta1})$
Inverse of logit, dependent on WEIGHT and AGE, has random effect	$Cl = \text{logitinv}(\text{theta1} + \text{theta2} * \text{WEIGHT} + \text{theta3} * \text{AGE} + \text{eta1})$

---

**Tip** To simultaneously fit data from multiple dose levels, use a `CovariateModel` object as an input argument to `sbiofitmixed`, and omit the random effect (`eta`) from the `Expression` property in the `CovariateModel` object.

---

The `Expression` property must meet the following requirements:

- The expressions are valid MATLAB code.
- Each expression is linear with a transformation.
- There is exactly one expression for each parameter.
- In each expression, a covariate is used in at most one term.
- In each expression, there is at most one random effect (`eta`)
- Fixed effect (`theta`) and random effect (`eta`) names are unique within and across expressions. That is, each covariate has its own fixed effect.

---

**Tip** Use the `getCovariateData` on page 2-201 method to view the covariate data when writing equations for the `Expression` property of a `CovariateModel` object.

---



---

**Tip** Use the `verify` on page 2-516 method to check that the `Expression` property of a `CovariateModel` object meets the conditions described previously.

---

## Characteristics

Applies to	Object: <code>CovariateModel</code> on page 2-98
------------	--

---

Data type	Character vector or cell array of character vectors
Data values	<code>parameterName = relationship</code>
Access	Read/write

## See Also

CovariateModel on page 2-98 | `getCovariateData` on page 2-201 | `verify` on page 2-516

## Topics

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”

“Specify a Covariate Model”

## Expression (AbstractKineticLaw, KineticLaw)

Expression to determine reaction rate equation

### Description

The `Expression` property indicates the mathematical expression that is used to determine the `ReactionRate` on page 3-177 property of the reaction object. `Expression` is a reaction rate expression assigned by the kinetic law definition used by the reaction. The kinetic law being used is indicated by the property `KineticLawName` on page 3-113. You can configure `Expression` for user-defined kinetic laws, but not for built-in kinetic laws. `Expression` is read only for kinetic law objects.

---

**Note** If you set the `Expression` property to a reaction rate expression that is not continuous and differentiable, see “Using Events to Address Discontinuities in Rule and Reaction Rate Expressions” before simulating your model.

---

### Kinetic Law Definition

The *kinetic law definition* provides a mechanism for applying a specific rate law to multiple reactions. It acts as a mapping template for the reaction rate. The kinetic law is defined by a mathematical expression, (defined in the property `Expression`), and includes the species and parameter variables used in the expression. The species variables are defined in the `SpeciesVariables` on page 3-217 property, and the parameter variables are defined in the `ParameterVariables` on page 3-158 property of the kinetic law object.

If a reaction is using a kinetic law definition, the `ReactionRate` property of the reaction object shows the result of a mapping from the kinetic law definition. To determine `ReactionRate`, the species variables and parameter variables that participate in the reaction rate should be mapped in the kinetic law for the reaction. In this case, SimBiology software determines the `ReactionRate` by using the `Expression` property of the abstract kinetic law object, and by mapping `SpeciesVariableNames` on page 3-215 to `SpeciesVariables` and `ParameterVariableNames` on page 3-156 to `ParameterVariables`.



For example, the kinetic law definition Henri-Michaelis-Menten has the Expression  $V_m * S / (K_m + S)$ , where  $V_m$  and  $K_m$  are defined as parameters in the ParameterVariables property of the abstract kinetic law object, and  $S$  is defined as a species in the SpeciesVariable property of the abstract kinetic law object.

By applying the Henri-Michaelis-Menten kinetic law to a reaction  $A \rightarrow B$  with  $V_a$  mapping to  $V_m$ ,  $A$  mapping to  $S$ , and  $K_a$  mapping to  $K_m$ , the rate equation for the reaction becomes  $V_a * A / (K_a + A)$ .

The exact expression of a reaction using MassAction kinetic law varies depending upon the number of reactants. Thus, for mass action kinetics the Expression property is set to MassAction because in general for mass action kinetics the reaction rate is defined as

$$r = k \prod_{i=1}^{n_r} [S_i]^{m_i}$$

where  $[S_i]$  is the concentration of the  $i$ th reactant,  $m_i$  is the stoichiometric coefficient of  $[S_i]$ ,  $n_r$  is the number of reactants, and  $k$  is the mass action reaction rate constant.

SimBiology software contains some built-in kinetic laws. You can also define your own kinetic laws. To find the list of available kinetic laws, use the `sbiowhos -kineticlaw` command (`sbiowhos` on page 1-340). You can create a kinetic law definition with the function `sbioabstractkineticlaw` and add it to the library using `sbioaddtolibrary` on page 1-22.

## Characteristics

Applies to	Objects: abstract kinetic law, kinetic law
Data type	Character vector
Data values	Defined by kinetic law definition
Access	Read-only in kinetic law object. Read/write in user-defined kinetic law.

## Examples

### Example 1

Example with Henri-Michaelis-Menten kinetics

- 1 Create a model object, and add a reaction object to the model.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Define a kinetic law for the reaction object.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

- 3 Verify that the Expression property for the kinetic law object is Henri-Michaelis-Menten.

```
get (kineticlawObj, 'Expression')
```

MATLAB returns:

```
ans =
```

```
Vm*S/(Km + S)
```

- 4 The 'Henri-Michaelis-Menten' kinetic law has two parameter variables (Vm and Km) and one species variable (S) that you should set. To set these variables, first create the parameter variables as parameter objects (parameterObj1, parameterObj2) with names Vm\_d, Km\_d, and assign the objects' Parent property value to the kineticlawObj. The species object with Name a is created when reactionObj is created and need not be redefined.

```
parameterObj1 = addparameter(kineticlawObj, 'Vm_d');  
parameterObj2 = addparameter(kineticlawObj, 'Km_d');
```

- 5 Set the variable names for the kinetic law object.

```
set(kineticlawObj, 'ParameterVariableNames', {'Vm_d' 'Km_d'});  
set(kineticlawObj, 'SpeciesVariableNames', {'a'});
```

- 6 Verify that the reaction rate is expressed correctly in the reaction object ReactionRate property.

```
get (reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =  
Vm_d*a/(Km_d+a)
```

## Example 2

Example with Mass Action kinetics.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Define a kinetic law for the reaction object.

```
kineticlawObj = addkineticlaw(reactionObj, 'MassAction');  
get(kineticlawObj, 'Expression')
```

MATLAB returns:

```
ans =  
MassAction
```

- 3 Assign the rate constant for the reaction.

```
set (kineticlawObj, 'ParameterVariableNames', 'k');  
get (reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =  
k*a*b
```

## See Also

KineticLawName, Parameters, ParameterVariableNames, ParameterVariables, ReactionRate, sbioaddtolibrary, sbiowhos, SpeciesVariables, SpeciesVariableNames

## FixedEffectDescription (CovariateModel)

Descriptions of fixed effects in CovariateModel object

### Description

The FixedEffectDescription property is a cell array of character vectors describing the fixed effects in the Expression on page 3-85 property of a CovariateModel object. Each character vector describes the role of a fixed effect in the expression equation. For example, in the following expression equation:

$$Cl = \exp(\text{theta1} + \text{theta2} * \text{WEIGHT} + \text{theta3} * \text{AGE} + \text{eta1})$$

The description for the fixed effect theta1 is 'Cl', which indicates it is the intercept for the parameter Cl. Also, the description for the fixed effect theta2 is 'Cl/WEIGHT', which indicates it is the slope of the line that defines the relationship between the parameter Cl and the covariate WEIGHT.

### Characteristics

Applies to	Object: CovariateModel on page 2-98
Data type	Cell array of character vectors
Data values	Description of the roles of the fixed effects in the Expression on page 3-85 property
Access	Read only

### See Also

CovariateModel on page 2-98 | Expression on page 3-85 | FixedEffectNames on page 3-93 | FixedEffectValues on page 3-94

## FixedEffectNames (CovariateModel)

Names of fixed effects in CovariateModel object

### Description

The FixedEffectNames property is a cell array of character vectors specifying the names of the fixed effects in the Expression on page 3-85 property of a CovariateModel object. Names of fixed effects are denoted with the prefix theta.

### Characteristics

Applies to	Object: CovariateModel on page 2-98
Data type	Cell array of character vectors
Data values	Names of the fixed effects in the Expression on page 3-85 property. These name are denoted with the prefix theta.
Access	Read only

### See Also

CovariateModel on page 2-98 | Expression on page 3-85 |  
FixedEffectDescription on page 3-92 | FixedEffectValues on page 3-94

## FixedEffectValues (CovariateModel)

Values for initial estimates of fixed effects in CovariateModel object

### Description

The FixedEffectValues property is a structure containing one field for each fixed effect in the Expression on page 3-85 property of a CovariateModel object. Each field contains the value of the initial estimate for a fixed effect.

---

**Tip** You must set this property before using the CovariateModel object as input to sbionlmeFit or sbionlmeFitSA. Use the constructDefaultFixedEffectValues on page 2-94 method to create a structure of fixed-effect initial estimate values, set to a default of zero. Then edit the structure and use it to modify this property.

---

### Characteristics

Applies to	Object: CovariateModel on page 2-98
Data type	Structure with one field for each fixed effect
Data values	Each field contains a double specifying the value of the initial estimate for a fixed effect in the CovariateModel object
Access	Read/write

### See Also

CovariateModel on page 2-98 | Expression on page 3-85 |  
FixedEffectDescription on page 3-92 | FixedEffectNames on page 3-93 |  
constructDefaultFixedEffectValues on page 2-94

### Topics

“Modeling the Population Pharmacokinetics of Phenobarbital in Neonates”

“Specify a Covariate Model”

## GroupID

Integer identifying each group in data set

### Description

GroupID is a property of the PKData object. It is an array of the same length as the DataSet on page 3-52 property containing an integer to identify each group. PKData sets this property during construction of the PKData object.

### Characteristics

Applies to	Object: PKData
Data type	double
Data values	Index value for each group
Access	Read-only

### See Also

PKData object



# GroupLabel

Identify group column in data set

## Description

GroupLabel is a property of the PKData object. It specifies the column in DataSet on page 3-52 that contains the group identification labels.

## Characteristics

Applies to	Object: PKData
Data type	Character vector
Data values	Column header from imported data set
Access	Read/write

## See Also

PKData object, GroupNames

## GroupNames

Unique values from GroupLabel in data set

### Description

GroupNames is a property of the PKData object. It contains unique values from the data column specified by the GroupLabel property. PKData sets this property during construction of the PKData object.

### Characteristics

Applies to	Object: PKData
Data type	Character vector or cell array of character vectors
Data values	Unique values in GroupLabel
Access	Read-only

### See Also

PKData object, GroupLabel

## HasLag

Lag associated with dose targeting compartment

### Description

HasLag is a property of the `PKCompartment` object. It is a logical indicating if the dose targeting the compartment has a time lag or not.

### Characteristics

Applies to	Object: <code>PKCompartment</code>
Data type	logical
Data values	1 (true) or 0 (false). Default is 0 (false).
Access	Read/write

### See Also

`addCompartment`, `DosingType`, `EliminationType`, `PKCompartment` object

## HasResponseVariable

Compartment drug concentration reported

### Description

`HasResponseVariable` is a property of the `PKCompartment` object. It is a logical indicating if the drug concentration in this compartment is reported.

---

**Note** The `HasResponseVariable` property can be `true` for more than one `PKCompartment` object in the model. If you perform a parameter fit on a model, at least one `PKCompartment` object in the model must have a `HasResponseVariable` property set to `true`.

---

### Characteristics

Applies to	Object: <code>PKCompartment</code>
Data type	Logical
Data values	1 (true) or 0 (false). Default is 0 (false).
Access	Read/write

### See Also

`addCompartment`, `DosingType`, `EliminationType`, `PKCompartment` object

# IndependentVarLabel

Identify independent variable column in data set

## Description

IndependentVarLabel is a property of the PKData object. It specifies the column in DataSet on page 3-52 that contains the independent variable (for example, time).

The column must contain positive values, and cannot contain, NaN, Inf or -Inf.

## Characteristics

Applies to	Object: PKData
Data type	Character vector
Data values	Column header from imported data set
Access	Read/write

## See Also

PKData object

## IndependentVarUnits

Time units in PKData object

### Description

The `IndependentVarUnits` property indicates the units for the column containing the independent variable (time) in the `PKData` object. If unit conversion is on, plot results in the SimBiology desktop show the units specified in `IndependentVarUnits`.

To get a list of units, use the `sbioshowunits` on page 1-286 function.

### Characteristics

Applies to	Object: PKData
Data type	Character vector
Data values	Time units. Default is '' (empty).
Access	Read/write

### See Also

`DependentVarLabel`, `PKData` object

# InitialAmount

Species initial amount

## Description

The `InitialAmount` property indicates the initial quantity of the SimBiology species object. `InitialAmount` is the quantity of the species before the simulation starts.

## Characteristics

Applies to	Object: species
Data type	double
Data values	Positive real number. Default value is 0.
Access	Read/write

## Examples

Add a species to a model and set the initial amount of the species.

- 1 Create a model object named `my_model`.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add the species object named `glucose`.

```
speciesObj = addspecies (modelObj, 'glucose');
```

- 3 Set the initial amount to 100 and verify.

```
set (speciesObj, 'InitialAmount', 100);  
get (speciesObj, 'InitialAmount')
```

MATLAB returns:

```
ans =  
    100
```

## **See Also**

`addspecies`, `InitialAmountUnits`



# InitialAmountUnits

Species initial amount units

## Description

The `InitialAmountUnits` property indicates the unit definition for the `InitialAmount` property of a species object. `InitialAmountUnits` can be one of the built-in units. To get a list of the defined units, use the `sbioshowunits` on page 1-286 function. If `InitialAmountUnits` changes from one unit definition to another, `InitialAmount` does not automatically convert to the new units. The `sbioconvertunits` function does this conversion. To add a user-defined unit to the list, use `sbiounit` followed by `sbioaddtolibrary`.

See `DefaultSpeciesDimension` for more information on specifying dimensions for species quantities. `InitialAmountUnits` must have corresponding dimensions to `CapacityUnits`. For example, if the `CapacityUnits` are  $\text{meter}^2$ , then species must be  $\text{amount}/\text{meter}^2$  or `amount`.

## Characteristics

Applies to	Object: species
Data type	Character vector
Data values	Units from library with dimensions of amount, amount/length, amount/area, or amount/volume. Default is '' (empty).
Access	Read/write

**Note** SimBiology uses units including empty units in association with `DimensionalAnalysis` and `UnitConversion` features.

- When `DimensionalAnalysis` and `UnitConversion` are both `false`, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.

- When `DimensionalAnalysis` is `true` and `UnitConversion` is `false`, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.
  - When `UnitConversion` is set to `true` (which requires `DimensionalAnalysis` to be `true`), SimBiology performs a dimensional analysis and converts everything to consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its unit to `dimensionless`.
- 

## Examples

- 1 Create a model object named `my_model`.

```
modelObj = sbiomodel ('my_model');  
compObj = addcompartment(modelObj, 'cell');
```

- 2 Add a species object named `glucose`.

```
speciesObj = addspecies (compObj, 'glucose');
```

- 3 Set the initial amount to 100, `InitialAmountUnits` to `molecule`, and verify.

```
set (speciesObj, 'InitialAmountUnits', 'molecule');  
get (speciesObj, 'InitialAmountUnits')
```

MATLAB returns:

```
ans =
```

```
molecule
```

## See Also

`DefaultSpeciesDimension`, `InitialAmount`, `sbioaddtolibrary`, `sbioconvertunits`, `sbioshowunits`, `sbiounit`,

## Inputs

Specify species and parameter input factors for sensitivity analysis

## Description

Inputs is a property of the `SensitivityAnalysisOptions` object. `SensitivityAnalysisOptions` is a property of the configuration set object.

Use `Inputs` to specify the species, parameters, or compartments with respect to which you want to compute the sensitivities of the species or parameter states in your model.

The SimBiology software calculates sensitivities with respect to the values of the parameters, capacities of compartments, and the initial amounts of the species specified in the `Inputs` property. When you simulate a model with `SensitivityAnalysis` enabled in the active configuration set object, sensitivity analysis returns the computed sensitivities of the species and parameters specified in the `Outputs` property. For a description of the output, see the `SensitivityAnalysisOptions` property description.

## Characteristics

Applies to	Object: <code>SensitivityAnalysisOptions</code>
------------	---

Data type	<p>Species, parameter, or compartment object or an array of objects</p> <hr/> <p><b>Note</b></p> <ul style="list-style-type: none"> <li>• If this object is determined by a repeated assignment rule, then you cannot use it as an <code>Inputs</code> property.</li> <li>• To be an input factor, a compartment object must have a constant capacity, that is, its <code>ConstantCapacity</code> property must be set to <code>true</code>.</li> <li>• If a parameter is referenced by the <code>LagParameterName</code> and <code>DurationParameterName</code> property of a <code>RepeatDose</code> object, the parameter must be constant to be an input for sensitivity analysis.</li> <li>• If a parameter is referenced by any other <code>RepeatDose</code> object properties, namely, <code>Amount</code>, <code>Rate</code>, <code>Interval</code>, <code>StartTime</code>, and <code>RepeatCount</code>, you cannot use the parameter as an input for sensitivity analysis.</li> </ul>
Data values	Species or parameter object, or an array of objects. Default is <code>[]</code> (empty array).
Access	Read/write

## Examples

This example shows how to set `Inputs` for sensitivity analysis.

- 1 Import the radio decay model from the SimBiology demos.
 

```
modelObj = sbmlimport('radiodecay');
```
- 2 Retrieve the configuration set object from `modelObj`.
 

```
configsetObj = getconfigset(modelObj);
```
- 3 Add a parameter to the `Inputs` property and display it. Use the `sbioselect` function to retrieve the parameter object from the model.
 

```
SimBiology Parameter Array
```

Index:	Name:	Value:	ValueUnits:
1	c	0.5	1/second

## See Also

Outputs, sbioselect, SensitivityAnalysis, SensitivityAnalysisOptions

## Interval

Time between doses

### Description

Interval is a property of a RepeatDose object. This property defines the equally spaced times between repeated doses.

For RepeatDose objects, you can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

---

**Note** When the Interval property is 0, RepeatDose ignores the RepeatCount property, that is, it treats it as though it is set to 0.

---

### Characteristics

Applies to	Object: RepeatDose.
Data type	double or character vector.
Data values	Nonnegative real number or name of a model-scoped parameter object. The default value is 0.
Access	Read/Write.

### See Also

RepeatDose object | ScheduleDose object

### Topics

“Parameterized and Adaptive Doses”

## KineticLaw

Show kinetic law used for ReactionRate

### Description

The KineticLaw property shows the kinetic law that determines the reaction rate specified in the ReactionRate property of the reaction object. This property shows the kinetic law used to define ReactionRate.

KineticLaw can be configured with the addkineticlaw on page 2-34 method. The addkineticlaw function configures the ReactionRate based on the KineticLaw and the species and parameters specified in the kinetic law object properties SpeciesVariableNames on page 3-215 and ParameterVariableNames on page 3-156. SpeciesVariableNames are determined automatically for mass action kinetics.

If you update the reaction, the ReactionRate property automatically updates only for mass action kinetics. For all other kinetics, you must set the SpeciesVariableNames property of the kinetic law object.

For information on dimensional analysis for reaction rates, see “How Reaction Rates Are Evaluated”.

### Characteristics

Applies to	Object: reaction
Data type	Kinetic law object
Data values	Kinetic law object. Default is [ ] (empty).
Access	Read-only

### Examples

Example with Henri-Michaelis-Menten kinetics

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Define a kinetic law for the reaction object.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

- 3 Verify that the KineticLaw property for the reaction object is Henri-Michaelis-Menten.

```
get (reactionObj, 'KineticLaw')
```

MATLAB returns:

SimBiology Kinetic Law Array

Index:	KineticLawName:
1	Henri-Michaelis-Menten

## See Also

KineticLawName, Parameters, ParameterVariableNames, ReactionRate, SpeciesVariableNames



# KineticLawName

Name of kinetic law applied to reaction

## Description

The `KineticLawName` property of the kinetic law object indicates the name of the kinetic law definition applied to the reaction. `KineticLawName` can be any valid name from the built-in or user-defined kinetic law library. See “Kinetic Law Definition” on page 3-88 for more information.

You can find the `KineticLawName` list in the kinetic law library by using the command `sbiowhos -kineticlaw` (`sbiowhos` on page 1-340). You can create a kinetic law definition with the function `sbioabstractkineticlaw` and add it to the library using `sbioaddtolibrary` on page 1-22.

## Characteristics

Applies to	Object: kineticlaw
Data type	Character vector
Data values	Character vector specified by kinetic law definition
Access	Read-only

## Examples

- 1 Create a model object, add a reaction object, and define a kinetic law for the reaction object.

```
modelObj = sbiomodel ('my_model');
reactionObj = addreaction (modelObj, 'a + b -> c + d');
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

- 2 Verify the `KineticLawName` of `kineticlawObj`.

```
get (kineticlawObj, 'KineticLawName')
```

MATLAB returns:

ans =

Henri-Michaelis-Menten

## See Also

Expression(AbstractKineticLaw, KineticLaw), Parameters, ParameterVariableNames, ParameterVariables, ReactionRate, sbioaddtolibrary, sbiowhos, SpeciesVariables, SpeciesVariableNames

# LagParameter

Parameter specifying time lag for doses

## Description

LagParameter is a property of the PKModelMap object. It specifies the name(s) of parameter object(s) that represent the time lag(s) of doses associated with the PKModelMap object.

Specify the name(s) of parameter object(s) that are:

- Scoped to a model
- Constant, that is, their ConstantValue property is true

When dosing multiple compartments, a one-to-one relationship must exist between the number and order of elements in the LagParameter property and the DosingType property. For a dose that has no lag, use '' (an empty character vector).

## Characteristics

Applies to	Object: PKModelMap
Data type	Character vector or cell array of character vectors
	<b>Tip</b> If you are not using any doses with time lags, you can set this property to a cell array of empty character vectors, or simply an empty cell array.

Data values	Name(s) of parameter object(s) or empty. Default is an empty cell array.  The parameter objects must be: <ul style="list-style-type: none"><li>• Scoped to a model</li><li>• Constant, that is, have a <code>ConstantValue</code> property set to <code>true</code>.</li></ul>
Access	Read/write

## See Also

DosingType, PKModelMap object

# LagParameterName

Parameter specifying time lag for dose

## Description

LagParameterName is a property of a RepeatDose or ScheduleDose object.

Specify the name of a parameter object that is scoped to a model. The parameter defines the length of time it takes for the dose to reach its target after being introduced.

You can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

## Characteristics

Applies to	Objects: RepeatDose, ScheduleDose.
Data type	Character vector.
Data values	Name of a model-scoped parameter object. The default value is an empty character vector ''.
Access	Read/write.

## Examples

### Estimate Time Lag and Duration of a Dose

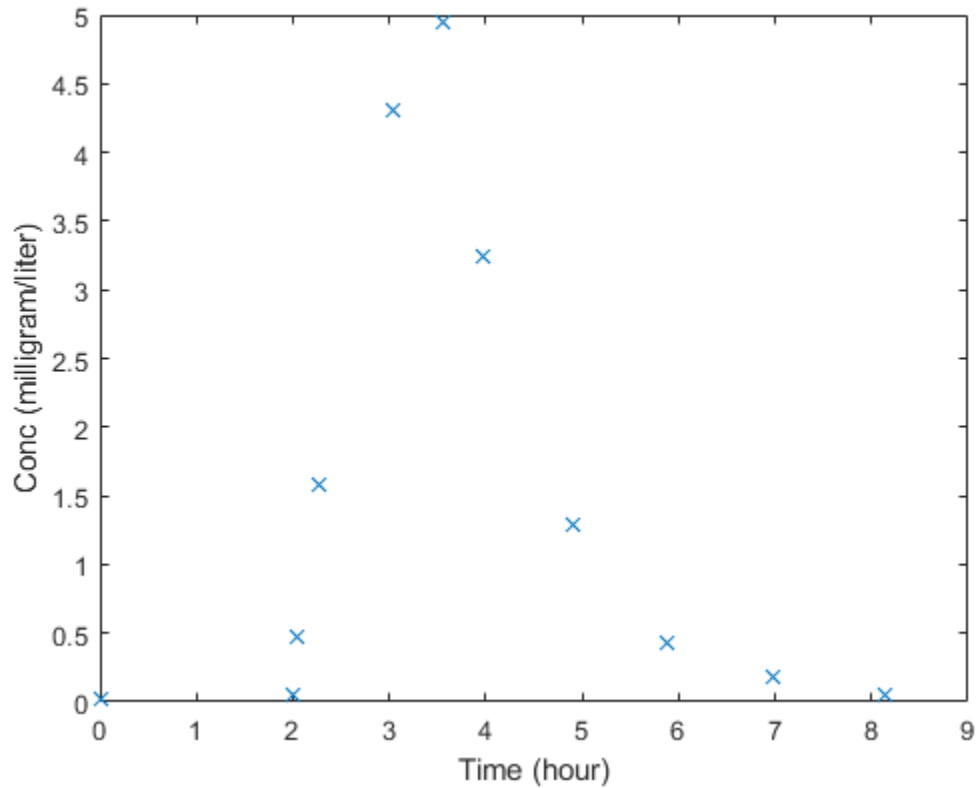
This example shows how to estimate the time lag before a bolus dose was administered and the duration of the dose using a one-compartment model.

Load a sample data set.

```
load lagDurationData.mat
```

Plot the data.

```
plot(data.Time,data.Conc,'x')  
xlabel('Time (hour)')  
ylabel('Conc (milligram/liter)')
```



Convert to groupedData.

```
gData = groupedData(data);  
gData.Properties.VariableUnits = {'hour','milligram/liter'};
```

Create a one-compartment model.

```
pkmd = PKModelDesign;  
pkc1 = addCompartment(pkmd,'Central');
```

```

pkc1.DosingType      = 'Bolus';
pkc1.EliminationType = 'linear-clearance';
pkc1.HasResponseVariable = true;
model                = construct(pkmd);
configset            = getconfigset(model);
configset.CompileOptions.UnitConversion = true;

```

Add two parameters that represent the time lag and duration of a dose. The lag parameter specifies the time lag before the dose is administered. The duration parameter specifies the length of time it takes to administer a dose.

```

lagP = addparameter(model, 'lagP');
lagP.ValueUnits = 'hour';
durP = addparameter(model, 'durP');
durP.ValueUnits = 'hour';

```

Create a dose object. Set the `LagParameterName` and `DurationParameterName` properties of the dose to the names of the lag and duration parameters, respectively.

```

dose          = sbiodose('dose');
dose.TargetName = 'Drug_Central';
dose.StartTime = 0;
dose.Amount    = 10;
dose.AmountUnits = 'milligram';
dose.TimeUnits = 'hour';
dose.LagParameterName = 'lagP';
dose.DurationParameterName = 'durP';

```

Map the model species to the corresponding data.

```

responseMap = {'Drug_Central = Conc'};

```

Specify the lag and duration parameters as parameters to estimate. Log-transform the parameters. Initialize them to 2 and set the upper bound and lower bound.

```

paramsToEstimate = {'log(lagP)', 'log(durP)'};
estimatedParams  = estimatedInfo(paramsToEstimate, 'InitialValue', 2, 'Bounds', [1 5]);

```

Perform parameter estimation.

```

fitResults = sbiofit(model, gData, responseMap, estimatedParams, dose, 'fminsearch')

```

```

fitResults =
  OptimResults with properties:

```

```
ExitFlag: 1
Output: [1x1 struct]
GroupName: One group
Beta: [2x4 table]
ParameterEstimates: [2x4 table]
J: [11x2 double]
COVB: [2x2 double]
CovarianceMatrix: [2x2 double]
R: [11x1 double]
MSE: 0.0024
SSE: 0.0213
Weights: []
LogLikelihood: 18.7511
AIC: -33.5023
BIC: -32.7065
DFE: 9
DependentFiles: {1x2 cell}
EstimatedParameterNames: {'lagP' 'durP'}
ErrorModelInfo: [1x3 table]
EstimationFunction: 'fminsearch'
```

Display the result.

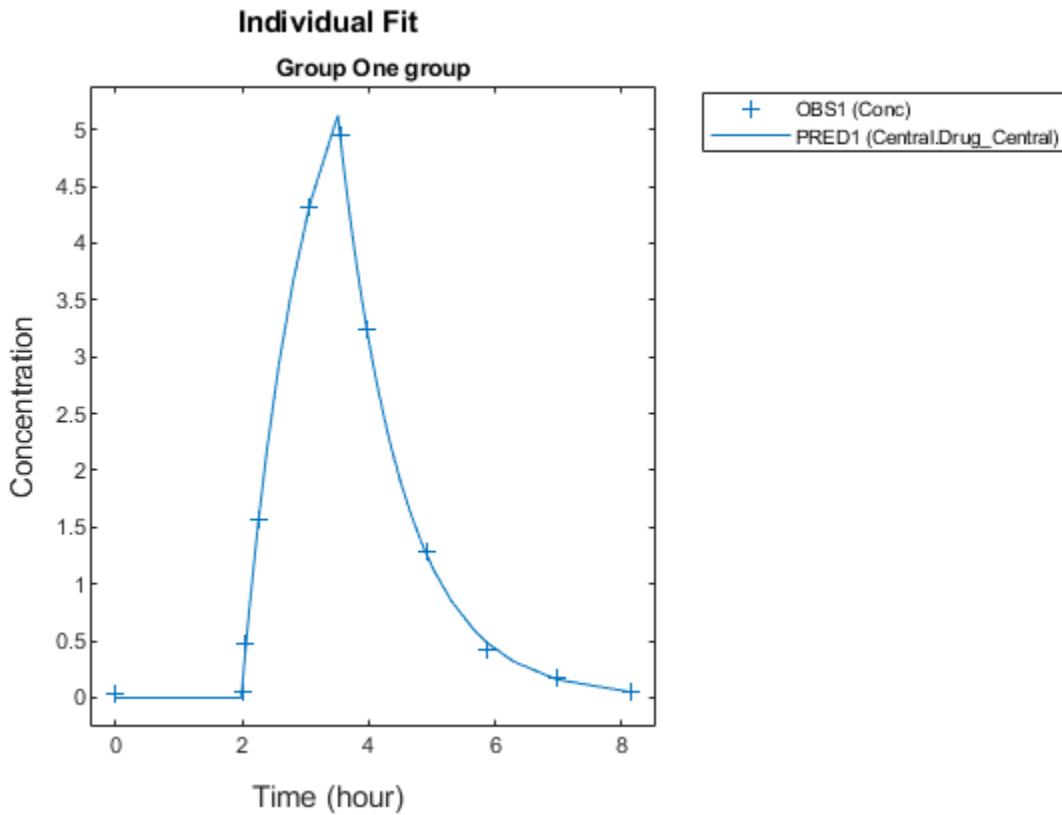
```
fitResults.ParameterEstimates
```

```
ans=2x4 table
```

Name	Estimate	StandardError	Bounds	
'lagP'	1.986	0.0051568	1	5
'durP'	1.527	0.012956	1	5

```
plot(fitResults)
```





## See Also

RepeatDose object | ScheduleDose object

## Topics

"Parameterized and Adaptive Doses"

## LogDecimation

Specify frequency to log stochastic simulation output

### Description

LogDecimation is a property of the SolverOptions property, which is a property of a configset object. This property defines how often stochastic simulation data is recorded. LogDecimation is available only for stochastic solvers (ssa, expltau, and impltau).

Use LogDecimation to specify how frequently you want to record the output of the simulation. For example, if you set LogDecimation to 1, for the command `[t,x] = sbiosimulate(modelObj)`, at each simulation step the time will be logged in `t` and the quantity of each logged species will be logged as a row in `x`. If LogDecimation is 10, then every 10th simulation step will be logged in `t` and `x`.

### Characteristics

Applies to	Object: SolverOptions
Data type	int
Data values	>0. Default is 1.
Access	Read/write

### Examples

This example shows how to change LogDecimation settings.

- 1 Retrieve the configset object from the modelObj, and change the SolverType to expltau.

```
modelObj = sbiomodel('cell');  
configsetObj = getconfigset(modelObj);  
set(configsetObj, 'SolverType', 'expltau')
```

- 2 Change the LogDecimation to 10.

```
set(configsetObj.SolverOptions, 'LogDecimation', 10);  
get(configsetObj.SolverOptions, 'LogDecimation')
```

```
ans =
```

```
10
```

## See Also

ErrorTolerance, RandomState

## MassUnits

Mass unit used internally during simulation when UnitConversion is on

### Description

This property defines the mass unit that SimBiology uses internally during model simulation when UnitConversion is on. You can set this to any string representing a mass unit such as gram, or gram with any valid prefix. It can also be a custom unit if it is consistent with mass as its dimension. The default is `<automatic>`, which means SimBiology automatically selects a mass unit for simulation. SimBiology examines the units on all of the states and selects a mass unit such that AbsoluteTolerance of the states in mass or mass per volume is at least as stringent as the simulation absolute tolerance multiplied by the smallest mass unit. This stringency is relaxed appropriately for states that become large when AbsoluteToleranceScaling is on.

---

**Note** It is recommended that you use the default unit (`<automatic>`) or choose units for states such that the simulated values are neither too large (greater than  $10^6$ ) or too small (less than  $10^{-6}$ ).

However, for some edge cases, you may need to change MassUnits. Suppose you have a model with a state that takes on values around  $10^{-12}$  gram for the entire simulation, and you need to use gram as its unit. Then it may be appropriate to set MassUnits to picogram. In this case, the internal simulation values would be around 1, instead of around  $10^{-12}$  as in the default case. AbsoluteTolerance of the simulation is determined using this internal value. Thus by choosing picogram as the mass unit, you effectively reduce the size of AbsoluteTolerance. Changing the MassUnits property is closely related to changing AbsoluteTolerance when considering the effects on simulation results.

Even when using the default unit, it may be still necessary to change AbsoluteTolerance. For details, see “Selecting Absolute Tolerance and Relative Tolerance for Simulation”.

If you need to recover the simulation behavior from releases prior to R2015b:

- Set the MassUnits to kilogram.

- Set the AmountUnits to mole. However, if the model has quantity units in molecule, set the unit to molecule instead.

---

**Tip** If you have a custom function and UnitConversion is on (whether or not you are using the default unit <automatic>), follow the recommendation below.

- Non-dimensionalize the parameters that are passed to the function if they are not already dimensionless.

Suppose you have a custom function defined as  $y = f(t)$  where  $t$  is the time in hour and  $y$  is the concentration of a species in mole/liter. When you use this function in your model to define a repeated assignment rule for instance, define it as:  $s1 = f(\text{time}/t0) * s0$ , where  $\text{time}$  is the simulation time,  $t0$  is a parameter defined as 1.0 hour,  $s0$  is a parameter defined as 1.0 mole/liter, and  $s1$  is the concentration of a species in mole/liter. Note that  $\text{time}$  and  $s1$  do not have to be in the same units as  $t0$  and  $s0$ , but they must be dimensionally consistent. For example, the  $\text{time}$  and  $s1$  units can be set to minute and picomole/liter, respectively.

---

## Characteristics

Applies to	Object: Configset
Data type	Character vector
Data values	Character vector specifying any mass unit. The default is <automatic>.
Access	Read/write for properties of Configset

## See Also

Configset object, AmountUnits

## MaximumNumberOfLogs

Maximum number of logs criteria to stop simulation

### Description

MaximumNumberOfLogs is a property of a Configset object. This property sets the maximum number of logs criteria to stop a simulation.

A simulation stops when it meets any of the criteria specified by StopTime, MaximumNumberOfLogs, or MaximumWallClock. However, if you specify the OutputTimes property of the SolverOptions property of the Configset object, then StopTime and MaximumNumberOfLogs are ignored. Instead, the last value in OutputTimes is used as the StopTime criteria, and the length of OutputTimes is used as the MaximumNumberOfLogs criteria.

### Characteristics

Applies to	Object: Configset
Data type	double
Data values	Positive value. Default is Inf.
Access	Read/write

### Examples

#### Set Maximum Number of Logs Criteria to Stop Simulation

Set the maximum number of logs that triggers a simulation to stop.

Create a model object named cell and save it in a variable named modelObj.

```
modelObj = sbiomodel('cell');
```

Retrieve the configuration set from `modelObj` and save it in a variable named `configsetObj`.

```
configsetObj = getconfigset(modelObj);
```

Configure the simulation stop criteria by setting the `MaximumNumberOfLogs` property to 50. Leave the `StopTime` and `MaximumWallClock` properties at their default values of 10 seconds and `Inf`, respectively.

```
set(configsetObj, 'MaximumNumberOfLogs', 50)
```

View the properties of `configsetObj`.

```
get(configsetObj)
```

```

        Active: 1
      CompileOptions: [1x1 SimBiology.CompileOptions]
             Name: 'default'
             Notes: ''
      RuntimeOptions: [1x1 SimBiology.RuntimeOptions]
SensitivityAnalysisOptions: [1x1 SimBiology.SensitivityAnalysisOptions]
      SolverOptions: [1x1 SimBiology.ODESolverOptions]
             SolverType: 'ode15s'
             StopTime: 10
MaximumNumberOfLogs: 50
      MaximumWallClock: Inf
             TimeUnits: 'second'
             AmountUnits: '<automatic>'
             MassUnits: '<automatic>'
             Type: 'configset'

```

When you simulate `modelObj`, the simulation stops when 50 logs are created or when the simulation time reaches 10 seconds, whichever comes first.

## See Also

`Configset` object, `MaximumWallClock`, `OutputTimes`, `StopTime`

## MaximumWallClock

Maximum elapsed wall clock time to stop simulation

### Description

MaximumWallClock is a property of a Configset object. This property sets the maximum elapsed wall clock time (seconds) criteria to stop a simulation.

A simulation stops when it meets any of the criteria specified by StopTime, MaximumNumberOfLogs, or MaximumWallClock. However, if you specify the OutputTimes property of the SolverOptions property of the Configset object, then StopTime and MaximumNumberOfLogs are ignored. Instead, the last value in OutputTimes is used as the StopTime criteria, and the length of OutputTimes is used as the MaximumNumberOfLogs criteria.

### Characteristics

Applies to	Object: Configset
Data type	double
Data values	Positive scalar. Default is Inf.
Access	Read/write

### Examples

#### Set Maximum Wall Clock Criteria to Stop Simulation

Set the maximum wall clock time (in seconds) that triggers a simulation to stop.

Create a model object named cell and save it in a variable named modelObj.

```
modelObj = sbiomodel('cell');
```



Retrieve the configuration set from `modelObj` and save it in a variable named `configsetObj`.

```
configsetObj = getconfigset(modelObj);
```

Configure the simulation stop criteria by setting the `MaximumWallClock` property to 20 seconds. Leave the `StopTime` and `MaximumNumberOfLogs` properties at their default values of 10 seconds and `Inf`, respectively.

```
set(configsetObj, 'MaximumWallClock', 20)
```

View the properties of `configsetObj`.

```
get(configsetObj)
```

```
Active: 1
CompileOptions: [1x1 SimBiology.CompileOptions]
Name: 'default'
Notes: ''
RuntimeOptions: [1x1 SimBiology.RuntimeOptions]
SensitivityAnalysisOptions: [1x1 SimBiology.SensitivityAnalysisOptions]
SolverOptions: [1x1 SimBiology.ODESolverOptions]
SolverType: 'ode15s'
StopTime: 10
MaximumNumberOfLogs: Inf
MaximumWallClock: 20
TimeUnits: 'second'
AmountUnits: '<automatic>'
MassUnits: '<automatic>'
Type: 'configset'
```

When you simulate `modelObj`, the simulation stops when the simulation time reaches 10 seconds or the wall clock time reaches 20 seconds, whichever comes first.

## See Also

`Configset` object, `MaximumNumberOfLogs`, `OutputTimes`, `StopTime`

## MaxIterations

Specify nonlinear solver maximum iterations in implicit tau

### Description

The `MaxIterations` property specifies the maximum number of iterations for the nonlinear solver in `impltau`. It is a property of the `SolverOptions` object. `SolverOptions` is a property of the `configset` object.

The implicit tau solver in SimBiology software internally uses a nonlinear solver to solve a set of algebraic nonlinear equations at every simulation step. Starting with an initial guess at the solution, the nonlinear solver iteratively tries to find the solution to the algebraic equations. The closer the initial guess is to the solution, the fewer the iterations the nonlinear solver will take before it finds a solution. `MaxIterations` specifies the maximum number of iterations the nonlinear solver should take before it issues a “failed to converge” error. If you get this error during simulation, try increasing `MaxIterations`. The default value of `MaxIterations` is 15.

### Characteristics

Applies to	Object: <code>SolverOptions</code>
Data type	<code>int</code>
Data values	>0. Default is 15.
Access	Read/write

### Examples

This example shows how to change `MaxIterations` settings.

- 1 Retrieve the `configset` object from the `modelObj`, and change the `SolverType` to `impltau`.

```
modelObj = sbiomodel('cell');  
configsetObj = getconfigset(modelObj);  
set(configsetObj, 'SolverType', 'impltau');
```

- 2 Change the MaxIterations to 25.

```
set(configsetObj.SolverOptions, 'MaxIterations', 25);  
get(configsetObj.SolverOptions, 'MaxIterations')
```

```
ans =
```

```
25
```

## See Also

ErrorTolerance, LogDecimation, RandomState

## MaxStep

Specify upper bound on ODE solver step size

### Description

MaxStep is a property of the SolverOptions property, which is a property of a configset object. This property specifies the bounds on the size of the time steps. MaxStep is available only for ODE solvers (ode15s, ode23t, ode45, and sundials).

If the differential equation has periodic coefficients or solutions, it might be a good idea to set MaxStep to some fraction (such as 1/4) of the period. This guarantees that the solver does not enlarge the time step too much and step over a period of interest. For more information on MaxStep, see odeset in the MATLAB documentation.

### Characteristics

Applies to	Object: SolverOptions
Data type	Positive scalar or empty
Data values	Default value is [] (empty), which is equivalent to setting MaxStep to infinity.
Access	Read/write

### See Also

SimBiology property RelativeTolerance

MATLAB function odeset

# ModelName

Name of model simulated

## Description

The ModelName property shows the name of the model for which the SimData object contains the simulation data.

## Characteristics

Applies to	Object: SimData
Data type	Character vector
Data values	Default value is ' ' (empty).
Access	Read-only

## See Also

Data, DataInfo

## Models

Contain all model objects

### Description

The `Models` property shows the models in the SimBiology root. It is a read-only array of model objects.

SimBiology has a hierarchical organization. A model object has the SimBiology root as its `Parent`. Parameter objects can have a model object or kinetic law object as `Parent`. You can display all the component objects with `modelObj.Models` or `get(modelObj, 'Models')`.

### Characteristics

Applies to	Objects: root
Data type	Array of model objects
Data values	Model object. Default is [ ] (empty).
Access	Read-only

### See Also

`sbiomodel`

# Multiplier

Relationship between defined unit and base unit

## Description

The `Multiplier` is the numerical value that defines the relationship between the unit Name and the base unit as a product of the `Multiplier` and the base unit. For example, in  $^{\circ}\text{Celsius} = (5/9) * (^{\circ}\text{Fahrenheit} - 32)$ ; `Multiplier` is 5/9 and `Offset` is -32. For 1 mole =  $6.0221e23 * \text{molecule}$ , the `Multiplier` is  $6.0221e23$ .

## Characteristics

Applies to	Object: Unit
Data type	double
Data values	Nonzero real number. Default value is 1.
Access	Read/write

## Examples

This example shows how to create a user-defined unit, add it to the user-defined library, and query the library.

- 1 Create a user-defined unit called `usermole`, whose composition is `molecule` and `Multiplier` property is  $6.0221e23$ .

```
unitObj = sbiounit('usermole', 'molecule', 6.0221e23);
```

- 2 Add the unit to the user-defined library.

```
sbioaddtolibrary(unitObj);
```

- 3 Query the `Multiplier` property.

```
get(unitObj, 'Multiplier')
```

ans =

1/molarity\*second

## **See Also**

Composition, get, Offset, sbiounit, set



## Name

Specify name of object

## Description

The Name property identifies a SimBiology object. Compartments, species, parameters, and model objects can be referenced by other objects using the Name property, therefore Name must be unique for these objects. However, species names need only be unique within each compartment. Parameter names must be unique within a model (if at the model level), or within each kinetic law (if at the kinetic law level). This means that you can have nonunique species names if the species are in different compartments, and nonunique parameter names if the parameters are in different kinetic laws or at different levels. Note that having nonunique parameter names can cause the model to have shadowed parameters and that may not be best modeling practice.

Use the function `sbioselect` to find an object with the same Name property value.

In addition, note the following constraints and reserved characters for the Name property in objects:

- Model and parameter names cannot be empty, the word `time`, all whitespace, or contain the characters `[` or `]`.
- Compartment and species names cannot be empty, the word `null`, the word `time` or contain the characters `->`, `<->`, `[` or `]`.
  - However, compartment and species names can contain the words `null` and `time` within the name, such as `nulldrug` or `nullreceptor`.
- Reaction, event, and rule names cannot be the word `time` or contain the characters `[` or `]`.
- If you have a parameter, a species, or compartment name that is not a valid MATLAB variable name, when you write an event function, an event trigger, a reaction, reaction rate equation, or a rule you must enclose that name in brackets. For example, enclose `[DNA polymerase+]` in brackets. In addition, if you have the same species in multiple compartments you must qualify the species with the compartment name, for example, `nucleus.[DNA polymerase+]`, `[nuclear complex].[DNA polymerase+]`.

For more information on valid MATLAB variable names, see `matlab.lang.makeValidName`, `matlab.lang.makeUniqueStrings`, and `isvarname`.

## Characteristics

Applies to	Objects: abstract kinetic law, configuration set, compartment, event, kinetic law, model, parameter, reaction, RepeatDose, rule, ScheduleDose, species, unit, or variant
Data type	Character vector
Data values	Any character vector except reserved words and characters
Access	Read/write

## Examples

- 1 Create a model object named `my_model`.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add a reaction object to the model object. Note the use of brackets because the names are not valid MATLAB variable names.

```
reactionObj = addreaction(modelObj, '[Aspartic acid] -> [beta-Aspartyl-P04]')
```

MATLAB returns:

```
SimBiology Reaction Array
```

```
Index:    Reaction:  
    1      [Aspartic acid] -> [beta-Aspartyl-P04]
```

- 3 Set the reaction Name and verify.

```
set (reactionObj, 'Name', 'Aspartate kinase reaction');  
get (reactionObj, 'Name')
```

MATLAB returns:

```
ans =
```

```
Aspartate kinase reaction
```

## See Also

addcompartment, addkineticlaw, addparameter, addreaction, addrule, addspecies, RepeatDose object, sbiomodel, sbiounit, sbiounitprefix, ScheduleDose object

## Normalization

Specify normalization type for sensitivity analysis

### Description

Normalization is a property of the `SensitivityAnalysisOptions` object. `SensitivityAnalysisOptions` is a property of the configuration set object. Use `Normalization` to specify the normalization for the computed sensitivities.

The following values let you specify the type of normalization. The examples show you how sensitivities of a species  $x$  with respect to a parameter  $k$  are calculated for each normalization type:

- 'None' specifies no normalization.

$$\frac{\partial x(t)}{\partial k}$$

- 'Half' specifies normalization relative to the numerator (species quantity) only.

$$\left(\frac{1}{x(t)}\right)\left(\frac{\partial x(t)}{\partial k}\right)$$

- 'Full' specifies that the data should be made dimensionless.

$$\left(\frac{k}{x(t)}\right)\left(\frac{\partial x(t)}{\partial k}\right)$$

### Characteristics

Applies to	Object: <code>SensitivityAnalysisOptions</code>
Data type	enum
Data values	'None', 'Half', 'Full'. Default is 'None'.

Access	Read/write
--------	------------

## See Also

Inputs, Outputs, SensitivityAnalysis, SensitivityAnalysisOptions

## Notes

HTML text describing SimBiology object

## Description

Use the Notes property of an object to store comments about the object. You can include HTML tagging in the notes to render formatted text in the SimBiology desktop.

## Characteristics

Applies to	Objects: compartment, kinetic law, model, parameter, reaction, RepeatDose, rule, ScheduleDose, species, unit, or unit prefix
Data type	Character vector
Data values	Any character vector
Access	Read/write

## Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Write notes for the model object.

```
set (modelObj, 'notes', '09/01/05 experimental data')
```

- 3 Verify the assignment.

```
get (modelObj, 'notes')
```

MATLAB returns:

```
ans =
```

```
09/01/05 experimental data
```

## See Also

addkineticlaw, addparameter, addreaction, addrule, addspecies, RepeatDose object, sbiomodel, sbiounit, sbiounitprefix, ScheduleDose object

## Observed

Measured response object name

## Description

Observed is a property of the `PKModelMap` object. It specifies the name(s) of one or more objects that represent the measured response (the response variable). Specify the name(s) of species or parameter object(s) that are scoped to a model.

## Characteristics

Applies to	Object: <code>PKModelMap</code>
Data type	Character vector or cell array of character vectors
Data values	Name of a species or parameter object or empty. Default is an empty cell array.
Access	Read/write

## See Also

Dosed, Estimated, `PKModelMap` object



# Offset

Unit composition modifier

## Description

---

**Note** The `Offset` property is currently not supported.

---

The `Offset` is the numerical value by which the unit composition is modified from the base unit. For example, `°Celsius = (5/9)*(°Fahrenheit-32)`; `Multiplier` is `5/9` and `Offset` is `-32`.

## Characteristics

Applies to	Object: Unit
Data type	double
Data values	Real number. Default is 0.
Access	Read/write

## Examples

This example shows how to create a user-defined unit, add it to the user-defined library, and query the library.

- 1 Create a user-defined unit called `celsius2`, whose composition refers to `fahrenheit`, `Multiplier` property is `9/5`, and `Offset` property is `32`.

```
unitObj = sbiounit('celsius2', 'fahrenheit', 9/5, 32);
```

- 2 Add the unit to the user-defined library.

```
sbioaddtolibrary(unitObj);
```

- 3 Query the `Offset` property.

```
get(unitObj, 'Offset')
```

```
ans =
```

```
32
```

### See Also

`Composition`, `get`, `Multiplier`, `sbioaddtolibrary`, `sbioshowunits`, `sbiounit`, `set`

## Outputs

Specify species and parameter outputs for sensitivity analysis

### Description

Outputs is a property of the `SensitivityAnalysisOptions` object. `SensitivityAnalysisOptions` is a property of the configuration set object.

Use `Outputs` to specify the species and parameters for which you want to compute sensitivities.

The SimBiology software calculates sensitivities with respect to the values of the parameters and the initial amounts of the species specified in the `Inputs` property. When you simulate a model with `SensitivityAnalysis` enabled in the active configuration set object, sensitivity analysis returns the computed sensitivities of the species and parameters specified in `Outputs`. For a description of the output, see the `SensitivityAnalysisOptions` property description.

### Characteristics

Applies to	Object: <code>SensitivityAnalysisOptions</code>
Data type	Species or parameter object or array of objects  <b>Note</b> If a species or parameter object is determined by a repeated assignment rule, then you cannot use it as an <code>Outputs</code> property.
Data values	Species or parameter object, or an array of objects. Default is [ ] (empty array).
Access	Read/write

## Examples

This example shows how to set `Outputs` for sensitivity analysis.

- 1 Import the radio decay model from the SimBiology demos.

```
modelObj = sbmlimport('radiodecay');
```

- 2 Retrieve the configuration set object from `modelObj`.

```
configsetObj = getconfigset(modelObj);
```

- 3 Add a species to the `Outputs` property and display it. Use the `sbioselect` function to retrieve the species object from the model.

```
SimBiology Species Array
```

```
Index:  Compartment:  Name:  InitialAmount:  InitialAmountUnits:
      1      unnamed      z      0      molecule
```

## See Also

`Inputs`, `sbioselect`, `SensitivityAnalysis`, `SensitivityAnalysisOptions`

# OutputTimes

Specify times to log deterministic simulation output

## Description

OutputTimes is a property of the SolverOptions property, which is a property of a Configset object. This property specifies the times during a deterministic (ODE) simulation that data is recorded. Time units are specified by the TimeUnits property of the Configset object. OutputTimes is available only for ODE solvers (ode15s, ode23t, ode45, and sundials).

If the criteria set in the MaximumWallClock property causes a simulation to stop before all time values in OutputTimes are reached, then no data is recorded for the latter time values.

The OutputTimes property can also control when a simulation stops:

- The last value in OutputTimes overrides the StopTime property as criteria for stopping a simulation.
- The length of OutputTimes overrides the MaximumNumberOfLogs property as criteria for stopping a simulation.

## Characteristics

Applies to	Object: SolverOptions
Data type	double
Data values	Vector of nonnegative, monotonically increasing values, or [], an empty vector. Default is [], which results in data being logged every time the simulation solver takes a step.
Access	Read/write

## Examples

### Specify Times to Log Deterministic Simulation Output

Specify the times during a deterministic (ODE) simulation that data is recorded.

Create a `model` object named `cell` and save it in a variable named `modelObj`.

```
modelObj = sbiomodel('cell');
```

Retrieve the configuration set from `modelObj` and save it in a variable named `configsetObj`.

```
configsetObj = getConfigset(modelObj);
```

Specify to log output every second for the first 10 seconds of the simulation. Do this by setting the `OutputTimes` property of the `SolverOptions` property of `ConfigsetObj`.

```
set(configsetObj.SolverOptions, 'OutputTimes', [1:10])  
get(configsetObj.SolverOptions, 'OutputTimes')
```

```
ans = 1×10
```

```
     1     2     3     4     5     6     7     8     9    10
```

When you simulate `modelObj`, output is logged every second for the first 10 seconds of the simulation. Also, the simulation stops after the 10th log.

## See Also

`MaximumNumberOfLogs`, `MaximumWallClock`, `SolverOptions`, `StopTime`, `TimeUnits`, `MassUnits`, `AmountUnits`

# Owner

Owning compartment

## Description

Owner shows you the SimBiology compartment object that owns the compartment object. In the compartment object, the Owner property shows you whether the compartment resides within another compartment. The Compartments property indicates whether other compartments reside within the compartment. You can add a compartment object using the method addcompartment.

## Characteristics

Applies to	Object: compartment
Data type	Character vector
Data values	Name of compartment object. Default is [].
Access	Read-only

## Examples

- 1 Create a model object named modelObj.

```
modelObj = sbiomodel('cell');
```

- 2 Add two compartments to the model object.

```
compartmentObj1 = addcompartment(modelObj, 'nucleus');
compartmentObj2 = addcompartment(modelObj, 'mitochondrion');
```

- 3 Add a compartment to one of the compartment objects.

```
compartmentObj3 = addcompartment(compartmentObj2, 'matrix');
```

- 4 Display the Owner property in the compartment objects.

```
get(compartmentObj3, 'Owner')
```

The result shows you the owning compartment and its components:

```
SimBiology Compartment - mitochondrion
```

```
Compartment Components:  
Capacity:          1  
CapacityUnits:  
Compartments:     1  
ConstantCapacity: true  
Owner:  
Species:          0
```

- 5 Change the owning compartment.

```
set(compartmentObj3, 'Owner', compartmentObj1)
```

## See Also

Compartments, Parent



## ParameterNames (CovariateModel)

Names of parameters in CovariateModel object

### Description

The ParameterNames property is a cell array of character vectors specifying the names of the parameters in the Expression on page 3-85 property of a CovariateModel object.

### Characteristics

Applies to	Object: CovariateModel on page 2-98
Data type	Cell array of character vectors
Data values	Names of the parameters in the Expression on page 3-85 property
Access	Read only

### See Also

CovariateModel on page 2-98 | Expression on page 3-85

## Parameters

Array of parameter objects

### Description

The `Parameters` property indicates the parameters in a `Model` or `KineticLaw` object. Read-only array of `Parameter` objects.

The scope of a parameter object is hierarchical and is defined by the parameter's parent. If a parameter is defined with a kinetic law object as its parent, then only the kinetic law object can use the parameter. If a parameter object is defined with a model object as its parent, then components such as rules, events, and kinetic laws (reaction rate equations) can use the parameter.

You can add a parameter to a model object, or kinetic law object with the method `addparameter` on page 2-40 and delete it with the method `delete` on page 2-95.

You can view parameter object properties with the `get` command and configure properties with the `set` command.

### Characteristics

Applies to	Objects: <code>model</code> , <code>kineticlaw</code>
Data type	Array of parameter objects
Data values	Parameter objects. Default value is <code>[]</code> (empty).
Access	Read-only

### Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Define a kinetic law for the reaction object.

```
kineticlawObj = addkineticlaw(reactionObj, 'MassAction');
```

- 3 Add a parameter and assign it to the kinetic law object (kineticlawObj);.

```
parameterObj1 = addparameter (kineticlawObj, 'K1');  
get (kineticlawObj, 'Parameters')
```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	K1	1	

- 4 Add a parameter and assign it to the model object (modelObj).

```
parameterObj1 = addparameter(modelObj, 'K2');  
get(modelObj, 'Parameters')
```

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	K2	1	

## See Also

addparameter, delete, get, set

## ParameterVariableNames

Cell array of reaction rate parameters

### Description

The `ParameterVariableNames` property shows the parameters used by the kinetic law object to determine the `ReactionRate` on page 3-177 equation in the reaction object. Use `setParameter` on page 2-449 to assign `ParameterVariableNames`. When you assign species to `ParameterVariableNames`, SimBiology software maps these parameter names to `ParameterVariables` on page 3-158 in the kinetic law object.

If the reaction is using a kinetic law, the `ReactionRate` property of a reaction object shows the result of a mapping from a “Kinetic Law Definition” on page 3-88. The `ReactionRate` is determined by the kinetic law object `Expression` property by mapping `ParameterVariableNames` to `ParameterVariables` and `SpeciesVariableNames` to `SpeciesVariables`.

### Characteristics

Applies to	Object: <code>kineticlaw</code>
Data type	Cell array of character vectors
Data values	Cell array of parameters
Access	Read/write

### Examples

Create a model, add a reaction, and assign the `SpeciesVariableNames` for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object of type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

reactionObj.KineticLaw property is configured to kineticlawObj.

- 3 The 'Henri-Michaelis-Menten' kinetic law has two parameter variables (Vm and Km) that should to be set. To set these variables:

```
setparameter(kineticlawObj, 'Vm', 'Va');  
setparameter(kineticlawObj, 'Km', 'Ka');
```

- 4 Verify that the parameter variables are correct.

```
get(kineticlawObj, 'ParameterVariableNames')
```

MATLAB returns:

```
ans =
```

```
    'Va'    'Ka'
```

## See Also

Expression(AbstractKineticLaw, KineticLaw), ParameterVariables, ReactionRate, setparameter, SpeciesVariables, SpeciesVariableNames

## ParameterVariables

Parameters in kinetic law definition

### Description

The `ParameterVariables` property shows the parameter variables that are used in the `Expression` on page 3-88 property of the abstract kinetic law object. Use this property to specify the parameters in the `ReactionRate` on page 3-177 equation. Use the method `set` to assign `ParameterVariables` to a kinetic law definition. For more information, see “Kinetic Law Definition” on page 3-88.

### Characteristics

Applies to	Objects: abstract kinetic law, kineticlaw
Data type	Cell array of character vectors
Data values	Specified by kinetic law definition
Access	Read/write in kinetic law definition. Read-only in kinetic law.

### Examples

Create a model, add a reaction, and assign the `SpeciesVariableNames` for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');  
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object of the type 'Henri-Michaelis-Menten' .

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

`reactionObj` `KineticLaw` property is configured to `kineticlawObj`.

- 3** The 'Henri-Michaelis-Menten' kinetic law has two parameter variables. To set these variables:

```
get (kineticlawObj, 'ParameterVariables')
```

MATLAB returns:

```
ans =
```

```
    'Vm'    'Km'
```

## See Also

Expression(AbstractKineticLaw, KineticLaw), ParameterVariableNames, ReactionRate, set, setparameter, SpeciesVariables, SpeciesVariableNames

## Parent

Indicate parent object

### Description

The **Parent** property indicates the parent object for a SimBiology object (read-only). The **Parent** property indicates accessibility of the object. The object is accessible to the **Parent** object and other objects within the **Parent** object. The value of **Parent** depends on the type of object and how it was created. All models always have the SimBiology root as the **Parent**.

### More Information

The following table shows you the different objects and the possible **Parent** value.

<b>Object</b>	<b>Parent</b>
abstract kinetic law	<ul style="list-style-type: none"> <li>• [ ] (empty) until added to library</li> <li>• root object upon addition to library</li> </ul>
compartment	model object
event	model object or [ ] (empty)
kinetic law	reaction object
model	root object
parameter	model object, kinetic law object, or [ ] (empty)
reaction	model object or [ ] (empty)
RepeatDose	model object or [ ] (empty)
rule	model object or [ ] (empty)
ScheduleDose	model object or [ ] (empty)
species	compartment
variant	model object or [ ] (empty)



Object	Parent
unit and unit prefixes	<ul style="list-style-type: none"> <li>• [ ] (empty) until added to library</li> <li>• root object upon addition to library</li> </ul>

## Characteristics

Applies to	Objects: abstract kinetic law, compartment, event, kinetic law, model, parameter, reaction, RepeatDose, rule, ScheduleDose, species, variant, unit, or unit prefix
Data type	Object
Data values	SimBiology component object or [ ] (empty)
Access	Read-only

## See Also

addkineticlaw, addparameter, addreaction, RepeatDose object, sbiomodel, ScheduleDose object

## PKCompartments

Hold compartments in PK model

### Description

PKCompartments is a property of the PKModelDesign object. It is used to specify the compartments in the PKModelDesign object. Each compartment is a PKCompartment object added using the addCompartment method.

### Characteristics

Applies to	Objects: PKModelDesign
Data type	object
Data values	PKCompartment object
Access	Read-only

### See Also

“Create Pharmacokinetic Models” in the SimBiology User's Guide, addCompartment, PKCompartment object, PKModelDesign object

# Products

Array of reaction products

## Description

The `Products` property contains an array of `SimBiology.Species` objects.

`Products` is a 1-by-n species object array that indicates the species that are changed by the reaction. If the `Reaction` property is modified to use a different species, the `Products` property is updated accordingly.

You can add product species to the reaction with `addproduct` on page 2-45 function. You can remove product species from the reaction with `rmproduct` on page 2-424. You can also update reaction products by setting the `Reaction` property with the function `set`.

## Characteristics

Applies to	Object: reaction
Data type	Array of objects
Data values	Species objects. Default is [] (empty).
Access	Read-only

## Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add reaction objects.

```
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 3 Verify the assignment.

```
productsObj = get(reactionObj, 'Products')
```

MATLAB returns:

SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	c	0	
2	unnamed	d	0	

## See Also

`addkineticlaw`, `addproduct`, `addspecies`, `rmproduct`

## RandomEffectNames (CovariateModel)

Names of random effects in CovariateModel object

### Description

The RandomEffectNames property is a cell array of character vectors specifying the names of the random effects in the Expression on page 3-85 property of a CovariateModel object. Names of random effects are denoted with the prefix eta.

### Characteristics

Applies to	Object: CovariateModel on page 2-98
Data type	Cell array of character vectors
Data values	Names of the random effects in the Expression on page 3-85 property. These name are denoted with the prefix eta.
Access	Read only

### See Also

CovariateModel on page 2-98 | Expression on page 3-85

## RandomState

Set random number generator

### Description

The `RandomState` property sets the random number generator for the stochastic solvers. It is a property of the `SolverOptions` object. `SolverOptions` is a property of the `configset` object.

SimBiology software uses a pseudorandom number generator. The sequence of numbers generated is determined by the state of the generator, which can be specified by the integer `RandomState`. If `RandomState` is set to integer  $J$ , the random number generator is initialized to its  $J^{\text{th}}$  state. The random number generator can generate all the floating-point numbers in the closed interval  $[2^{(-53)}, 1-2^{(-53)}]$ . Theoretically, it can generate over  $2^{1492}$  values before repeating itself. But for a given state, the sequence of numbers generated will be the same. To change the sequence, change `RandomState`. SimBiology software resets the state at startup. The default value of `RandomState` is `[]`.

### Characteristics

Applies to	Objects: <code>SolverOptions</code> for SSA, <code>expltau</code> , <code>impltau</code>
Data type	<code>int</code>
Data values	Default is <code>[]</code> (empty).
Access	Read/write

### Examples

This example shows how to change `RandomState` settings.

- 1 Retrieve the `configset` object from the `modelObj` and change the `SolverType` to `expltau`.

```
modelObj = sbiomodel('cell');  
configsetObj = getconfigset(modelObj);  
set(configsetObj, 'SolverType', 'expltau')
```

- 2 Change the Randomstate to 5.

```
set(configsetObj.SolverOptions, 'RandomState', 5);  
get(configsetObj.SolverOptions, 'RandomState')
```

```
ans =
```

```
5
```

## See Also

ErrorTolerance, LogDecimation, MaxIterations

## Rate

Rate of dose

## Description

Rate is a property of a `RepeatDose` or `ScheduleDose` object.

This property defines how fast a dose is given. If the rate is set to 0 or an empty array [], then it is interpreted as a bolus (instantaneous) dose.

For `RepeatDose` objects, you can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

---

**Note** If you set the `Rate` property of a dose, you must also specify the `Amount` property of the dose, and set the `DurationParameterName` property to ''. This is because the duration is calculated from the amount and rate.

---

---

**Tip** You can create a combination of bolus and infusion doses by setting the `rate` property of a `ScheduleDose` object to a vector containing zeros and non-zeros.

---

## Characteristics

Applies to	Objects: <code>RepeatDose</code> , <code>ScheduleDose</code> .
Data type	double or character vector ( <code>RepeatDose</code> ) or double array ( <code>ScheduleDose</code> ).
Data values	Nonnegative real number or name of a model-scoped parameter object. The default value is 0 ( <code>RepeatDose</code> ) or [] ( <code>ScheduleDose</code> ), that is, the dose is interpreted as a bolus (instantaneous) dose.
Access	Read/write.



## **See Also**

RepeatDose object | ScheduleDose object

## **Topics**

“Parameterized and Adaptive Doses”

## RateUnits

Units for dose rate

### Description

RateUnits is a property of a PKData, RepeatDose or ScheduleDose object.

- In RepeatDose or ScheduleDose objects, this property defines units for the Rate property.
- In PKData object, this property defines units for the RateLabel property.

### Characteristics

Applies to	Object: RepeatDose, ScheduleDose, PKData
Data type	Character vector
Data values	Units from library with dimensions of amount divided by time. You cannot use units of concentration divided by time. Default = "" (empty).
Access	Read/write

---

**Note** SimBiology uses units including empty units in association with DimensionalAnalysis and UnitConversion features.

- When DimensionalAnalysis and UnitConversion are both false, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.
- When DimensionalAnalysis is true and UnitConversion is false, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.
- When UnitConversion is set to true (which requires DimensionalAnalysis to be true), SimBiology performs a dimensional analysis and converts everything to

consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its units to dimensionless.

---

## **See Also**

PKData object, ScheduleDose object, RepeatDose object, Rate, RateLabel

## RateLabel

Rate of infusion column in data set

### Description

RateLabel is a property of the PKData object. It specifies the column in DataSet on page 3-52 that contains the rate of infusion. This applies only when dosing type is infusion. The data set must contain the rate and not an infusion time. The values must be positive and the column cannot contain Inf or -Inf. 0 specifies an infinite rate (equivalent to a bolus dose), and NaN specifies no rate.

### Characteristics

Applies to	Objects: PKData
Data type	Character vector
Data values	Column header
Access	Read/write

### See Also

PKData object, DosingType

# Reactants

Array of reaction reactants

## Description

The `Reactants` property is a 1-by-n species object array of reactants in the reaction. If the `Reaction` property is modified to use a different reactant, the `Reactants` property will be updated accordingly.

You can add reactant species to the reaction with the `addreactant` on page 2-48 method.

You can remove reactant species from the reaction with the `rmreactant` on page 2-426 method. You can also update reactants by setting the `Reaction` property with the function `set`.

## Characteristics

Applies to	Object: reaction
Data type	Species object or array of species objects
Data values	Species objects. Default is [] (empty).
Access	Read-only

## Examples

- 1 Create a model object.

```
modelObj = sbiomodel ('my_model');
```
- 2 Add reaction objects.

```
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```
- 3 View the reactants for `reactionObj`.

```
get(reactionObj, 'Reactants')
```

MATLAB returns:

SimBiology Species Array

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	a	0	
2	unnamed	b	0	

## See Also

`addreactant`, `addreaction`, `addspecies`, `rmreactant`

# Reaction

Reaction object reaction

## Description

Property to indicate the reaction represented in the reaction object. Indicates the chemical reaction that can change the amount of one or more species, for example, 'A + B -> C'. This property is different from the model object property called `Reactions`.

See `addreaction` for more information on how the `Reaction` property is set.

## Characteristics

Applies to	Object: reaction
Data type	Character vector
Data values	Character vector containing a valid reaction. Default is '' (empty character vector).
Access	Read/write

## Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Verify that the reaction property records the input.

```
get (reactionObj, 'Reaction')
```

MATLAB returns:

```
ans =
```

```
a + b -> c + d
```

## **See Also**

addreaction



## ReactionRate

Reaction rate equation in reaction object

### Description

The `ReactionRate` property defines the reaction rate equation. You can define a `ReactionRate` with or without the `KineticLaw` property. `KineticLaw` defines the type of reaction rate. The `addkineticlaw` function configures the `ReactionRate` based on the `KineticLaw` and the species and parameters specified in the kinetic law object properties `SpeciesVariableNames` and `ParameterVariableNames`.

The reaction takes place in the reverse direction if the `Reversible` property is true. This is reflected in `ReactionRate`. The `ReactionRate` includes the forward and reverse rate if reversible.

You can specify `ReactionRate` without `KineticLaw`. Use the `set` function to specify the reaction rate equation. SimBiology software adds species variables while creating `reactionObj` using the `addreaction` method. You must add the parameter variables (to the `modelObj` in this case). See the example below.

After you specify the `ReactionRate` without `KineticLaw` and you later configure the `reactionObj` to use `KineticLaw`, the `ReactionRate` is unset until you specify `SpeciesVariableNames` and `ParameterVariableNames`.

For information on dimensional analysis for reaction rates, see “How Reaction Rates Are Evaluated” .

---

**Note** If you set the `ReactionRate` property to an expression that is not continuous and differentiable, see “Using Events to Address Discontinuities in Rule and Reaction Rate Expressions” before simulating your model.

---

### Characteristics

Applies to	Object: reaction
------------	------------------

Data type	Character vector
Data values	Character vector defining the reaction rate. Default is ' ' (empty character vector).
Access	Read/write

## Examples

### Add a Reaction Defined by Michaelis-Menten Kinetic Law

Create a model, add a reaction, and assign the expression for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

reactionObj.KineticLaw property is configured to kineticlawObj.

- 3 The 'Henri-Michaelis-Menten' kinetic law has two parameter variables (Vm and Km) and one species variable (S) that you should set. To set these variables, first create the parameter variables as parameter objects (parameterObj1, parameterObj2) with names Vm\_d and Km\_d and assign them to kineticlawObj.

```
parameterObj1 = addparameter(kineticlawObj, 'Vm_d');
parameterObj2 = addparameter(kineticlawObj, 'Km_d');
```

- 4 Set the variable names for the kinetic law object.

```
set(kineticlawObj, 'ParameterVariableNames', {'Vm_d' 'Km_d'});
set(kineticlawObj, 'SpeciesVariableNames', {'a'});
```

- 5 Verify that the reaction rate is expressed correctly in the reaction object ReactionRate property.

```
get (reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =
Vm_d*a/(Km_d + a)
```

## Add a Reaction without a Kinetic Law

Create a model, add a reaction, and specify ReactionRate without a kinetic law.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a + b -> c + d');
```

- 2 Specify ReactionRate and verify the assignment.

```
set(reactionObj, 'ReactionRate', 'k*a');
get(reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =
k*a
```

- 3 You cannot simulate the model until you add the parameter k to the modelObj.

```
parameterObj = addparameter(modelObj, 'k');
```

SimBiology adds the parameter to the modelObj with default Value = 1.0 for the parameter.

## Define a Custom Hill Kinetic Law that Works with Dimensional Analysis

This example shows how to define a custom reaction rate for the Hill kinetics that is compatible with DimensionalAnalysis feature of SimBiology.

This example is useful especially if you are using the built-in Hill kinetic law, but have the kinetic reaction with a non-integer exponent and cannot verify the model because dimensional analysis failed. The built-in Hill kinetic law has the following expression:

$$\frac{V_m * S^n}{K_p + S^n}. \text{ Suppose } K_p = K_h^n, \text{ then you can rewrite the equation as follows: } \frac{V_m}{\left(\frac{K_h}{S}\right)^n + 1}.$$

The redefined Hill kinetic equation is compatible with Dimensional Analysis and allows you to have a non-integer exponent.

Create a SimBiology model.

```
m1 = sbiomodel('m1');
```

Add a compartment, two species, and a reaction.

```
c1 = addcompartment(m1, 'cell');  
s1 = addspecies(m1, 'a');  
s2 = addspecies(m1, 'b');  
r1 = addreaction(m1, 'a -> b');
```

Add a predefined a Hill kinetic law for the reaction.

```
k1 = addkineticlaw(r1, 'Hill-Kinetics');
```

Display the rate expression of the built-in kinetic law.

```
k1.Expression
```

```
ans =
```

```
Vm*S^n/(Kp + S^n)
```

Define parameters, values, and units.

```
p1 = addparameter(k1, 'Vm', 1.0);  
p2 = addparameter(k1, 'n', 1.5);  
p3 = addparameter(k1, 'Kp', 2.828);  
  
set(k1, 'ParameterVariableNames', {'Vm', 'n', 'Kp'});  
set(k1, 'SpeciesVariableNames', {'a'});  
set(s1, 'InitialAmount', 2.0);  
  
set(s1, 'InitialAmountUnits', 'mole/liter');  
set(s2, 'InitialAmountUnits', 'mole/liter');  
set(c1, 'CapacityUnits', 'liter');  
set(p1, 'ValueUnits', 'mole/liter/second');  
set(p2, 'ValueUnits', 'dimensionless');  
set(p3, 'ValueUnits', 'mole/liter');
```

Verify the model.

```
verify(m1)
```

```
Error using SimBiology.Model/verify
--> Error reported from Dimensional Analysis:
Dimensional analysis failed for reaction 'a -> b'.
When using the power function, both the base and exponent must be dimensionless or the
integer constant (for example 2 in 'x^2').
```

You are seeing the error message because SimBiology only allows exponentiation of any dimensionless quantity to any dimensionless power.

Redefine the reaction rate so that it is compatible with dimensional analysis and allows a non-integer exponent.

```
r1.ReactionRate = 'Vm / ( (Kh/a)^n + 1 )';
k1.KineticLaw = 'Unknown';
```

Define the value and units for Kh parameter.

```
p4 = addparameter(k1, 'Kh', 2.0);
set(p4, 'ValueUnits', 'mole/liter');
```

Verify the model.

```
verify(m1)
```

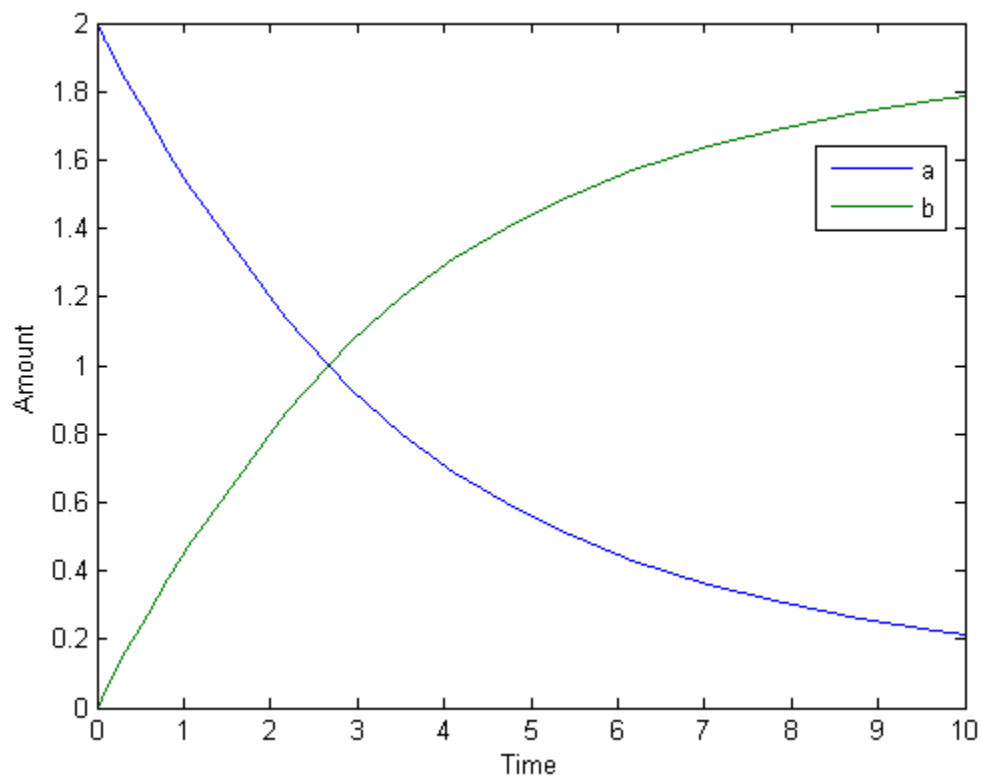
You no longer see the error message.

Simulate the model.

```
[t,x,names] = sbiosimulate(m1);
```

Plot the results.

```
plot(t,x);
xlabel('Time');
ylabel('Amount');
legend(names);
```



## See Also

addparameter, addreaction, Reversible

# Reactions

Array of reaction objects

## Description

Property to indicate the reactions in a `Model` object. Read-only array of reaction objects.

A reaction object defines a chemical reaction that occurs between species. The species for the reaction are defined in the `Model` object property `Species`.

You can add a reaction to a model object with the method `addreaction` on page 2-51, and you can remove a reaction from the model object with the method `delete` on page 2-95.

## Characteristics

Applies to	Object: model
Data type	Array of reaction objects
Data values	Reaction object
Access	Read-only

## Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Verify that the reactions property records the input.

```
get (modelObj, 'Reactions')
```

MATLAB returns:

SimBiology Reaction Array

Index:	Reaction:
1	a + b -> c + d

## See Also

addreaction, delete



# RelativeTolerance

Allowable error tolerance relative to state value during a simulation

## Description

RelativeTolerance is a property of the SolverOptions object, which is a property of a Configset object. It is available for the ode solvers (ode15s, ode23t, ode45, and sundials).

The RelativeTolerance property specifies the allowable error tolerance relative to the state vector at each simulation step. The state vector contains values for all the state variables, for example, amounts for all the species.

If you set the RelativeTolerance at  $1e-2$ , you are specifying that an error of 1% relative to each state value is acceptable at each simulation step.

For details, see “Selecting Absolute Tolerance and Relative Tolerance for Simulation”.

## Characteristics

Applies to	Object: SolverOptions
Data type	double
Data values	Positive scalar that is <1. Default is $1e-3$ .
Access	Read/write

## Examples

This example shows how to change AbsoluteTolerance.

- 1 Retrieve the configset object from the modelObj.

```
modelObj = sbiomodel('cell');
configsetObj = getConfigset(modelObj)
```

- 2 Change the `AbsoluteTolerance` to `1e-8`.

```
set(configsetObj.SolverOptions, 'RelativeTolerance', 1.0e-6);  
get(configsetObj.SolverOptions, 'RelativeTolerance')
```

```
ans =
```

```
1.0000e-006
```

## See Also

`AbsoluteTolerance`, `MassUnits`, `AmountUnits`, `Configset` object

## See Also

### Topics

“Model Simulation”

“Choosing a Simulation Solver”

“Ordinary Differential Equations” (MATLAB)

# RepeatCount

Dose repetitions

## Description

RepeatCount is a property of a RepeatDose object. This property defines the number of doses after the initial dose in a repeat dose series.

For RepeatDose objects, you can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

---

**Note** When the Interval property is 0, RepeatDose ignores the RepeatCount property, that is, it treats it as though it is set to 0.

---

## Characteristics

Applies to	Object: RepeatDose.
Data type	double or character vector.
Data values	Nonnegative integer or name of a model-scoped parameter object. The default value is 0.
Access	Read/Write.

## See Also

RepeatDose object | ScheduleDose object

## Topics

“Parameterized and Adaptive Doses”

## Reversible

Specify whether reaction is reversible or irreversible

### Description

The `Reversible` property defines whether a reaction is reversible or irreversible. The rate of the reaction is defined by the `ReactionRate` property. For a reversible reaction, the reaction rate equation is the sum of the rate of the forward and reverse reactions. The type of reaction rate is defined by the `KineticLaw` property. If a reaction is changed from reversible to irreversible or vice versa after `KineticLaw` is assigned, the new `ReactionRate` is determined only if `Type` is `MassAction`. All other `Types` result in unchanged `ReactionRate`. For `MassAction`, the first parameter specified is assumed to be the rate of the forward reaction.

### Characteristics

Applies to	Object: reaction
Data type	boolean
Data values	true, false. Default value is false.
Access	Read/write

### Examples

Create a model, add a reaction, and assign the expression for the reaction rate equation.

- 1 Create model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Set the `Reversible` property for the `reactionObj` to `true` and verify this setting.

```
set (reactionObj, 'Reversible', true)
get (reactionObj, 'Reversible')
```

MATLAB returns:

```
ans =
```

```
1
```

MATLAB returns 1 for true and 0 for false.

In the next steps the example illustrates how the reaction rate equation is assigned for reversible reactions.

- 3 Create a kinetic law object for the reaction object of the type 'MassAction'.

```
kineticlawObj = addkineticlaw(reactionObj, 'MassAction');
```

reactionObj KineticLaw property is configured to kineticlawObj.

- 4 The 'MassAction' kinetic law for reversible reactions has two parameter variables ('Forward Rate Parameter' and 'Reverse Rate Parameter') that you should set. The species variables for MassAction are automatically determined. To set the parameter variables, first create the parameter variables as parameter objects (parameterObj1, parameterObj2) named Kf and Kr and assign the object to kineticlawObj.

```
parameterObj1 = addparameter(kineticlawObj, 'Kf');
parameterObj2 = addparameter(kineticlawObj, 'Kr');
```

- 5 Set the variable names for the kinetic law object.

```
set(kineticlawObj, 'ParameterVariableNames', {'Kf' 'Kr'});
```

- 6 Verify that the reaction rate is expressed correctly in the reaction object ReactionRate property.

```
get (reactionObj, 'ReactionRate')
```

MATLAB returns:

```
ans =
```

```
Kf*a*b - Kr*c*d
```

## **See Also**

`addparameter`, `addreactant`, `addreaction`, `ParameterVariableNames`, `ReactionRate`

## Rule

Specify species and parameter interactions

### Description

The `Rule` property contains a rule that defines how certain species and parameters should interact with one another. For example, a rule could state that the total number of species A and species B must be some value. `Rule` is a MATLAB expression that defines the change in the species object quantity or a parameter object `Value` on page 3-247 when the rule is evaluated.

You can add a rule to a model object with the `addrule` on page 2-57 method and remove the rule with the `delete` on page 2-95 method. For more information on rules, see `addrule` on page 2-57 and `RuleType` on page 3-193.

---

**Note** If you set the `Rule` property for an algebraic rule, rate rule, or repeated assignment rule, and the rule expression is not continuous and differentiable, see “Using Events to Address Discontinuities in Rule and Reaction Rate Expressions” before simulating your model.

---

### Characteristics

Applies to	Object: rule
Data type	Character vector
Data values	Character vector defined as species or parameter objects. Default is an empty character vector ' '.
Access	Read/write

### Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

2 Add a rule.

```
ruleObj = addrule(modelObj, '10-a+b')
```

MATLAB returns:

SimBiology Rule Array

Index:	RuleType:	Rule:
1	algebraic	10-a+b

## See Also

`addrule`, `delete`, “Definitions and Evaluations of Rules”



# RuleType

Specify type of rule for rule object

## Description

The `RuleType` property indicates the type of rule defined by the rule object. A `Rule` object defines how certain species, parameters, and compartments should interact with one another. For example, a rule could state that the total number of species A and species B must be some value. `Rule` is a MATLAB expression that defines the change in the species object quantity or a parameter object `Value` on page 3-247 when the rule is evaluated.

You can add a rule to a model object with the `addrule` on page 2-57 method and remove the rule with the `delete` on page 2-95 method. For more information on rules, see `addrule` on page 2-57.

The types of rules in SimBiology are as follows:

- `initialAssignment` — Lets you specify the initial value of a parameter, species, or compartment capacity, as a function of other model component values in the model.
- `repeatedAssignment` — Lets you specify a value that holds at all times during simulation, and is a function of other model component values in the model.
- `algebraic` — Lets you specify mathematical constraints on one or more parameters, species, or compartments that must hold during a simulation.
- `rate` — Lets you specify the time derivative of a parameter value, species amount, or compartment capacity.

## Constraints on Varying Species Using a Rate Rule

If the model has a species defined in concentration, being varied by a `rate` rule, and it is in a compartment with varying volume, you can only use `rate` or `initialAssignment` rules to vary the compartment volume.

Conversely, if you are varying a compartment's volume using a `repeatedAssignment` or `algebraic` rules, then you cannot vary a species (defined in concentration) within that compartment, with a `rate` rule.

The reason for these constraints is that, if a species is defined in concentration and it is in a compartment with varying volume, the time derivative of that species is a function of the compartment's rate of change. For compartments varied by rate rules, the solver has that information.

Note that if you specify the species in amounts there are no constraints.

## Characteristics

Applies to	Object: rule
Data type	Character vector
Data values	'initialAssignment', 'repeatedAssignment' 'algebraic', 'rate'. Default value is 'initialAssignment'.
Access	Read/write

## Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');
reactionObj = addreaction (modelObj, 'a -> b');
```

- 2 Add a rule that specifies the quantity of a species c. In the rule expression, k is the rate constant for a -> b.

```
ruleObj = addrule(modelObj, 'c = k*(a+b)')
```

- 3 Change the RuleType from the default ('initialAssignment') to 'rate' and verify it using the get command.

```
set(ruleObj, 'RuleType', 'rate');
get(ruleObj)
```

MATLAB returns all the properties for the rule object.

```
Active: 1
Annotation: ''
Name: ''
Notes: ''
```

```
Parent: [1x1 SimBiology.Model]
Rule: 'c = k*(a+b) '
RuleType: 'rate'
Tag: ''
Type: 'rule'
UserData: []
```

## See Also

“Definitions and Evaluations of Rules”, `addrule`, `delete`

## Rules

Array of rules in model object

## Description

The Rules property shows the rules in a `Model` object. Read-only array of `SimBiology.Rule` objects.

A *rule* is a mathematical expression that modifies a species amount or a parameter value. A rule defines how certain species and parameters should interact with one another. For example, a rule could state that the total number of species A and species B must be some value.

You can add a rule to a model object with the `addrule` on page 2-57 method and remove the rule with the `delete` on page 2-95 method. For more information on rules, see `addrule` and `RuleType` on page 3-193.

## Characteristics

Applies to	Object: model
Data type	Array of rule objects
Data values	Rule object
Access	Read-only

## Examples

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');  
reactionObj = addreaction (modelObj, 'a + b -> c + d');
```

- 2 Add a rule.

```
ruleObj = addrule(modelObj, '10-a+b')
```

MATLAB returns:

SimBiology Rule Array

Index:	RuleType:	Rule:
1	algebraic	10-a+b

## See Also

addrule, delete, "Definitions and Evaluations of Rules"

## RunInfo

Information about simulation

### Description

The RunInfo property contains information describing the simulation run that yielded the data in the SimData object.

The following information is stored:

- **Configset** — A struct form of the configuration set used during simulation. This would typically be the model's active configset.
- **Variant** — A struct form of the variant(s) used during simulation.
- **SimulationDate** — The date/time of simulation.
- **SimulationType** — Either 'single run' or 'ensemble run', depending on whether the data object was created using the function `sbiosimulate` or the function `sbioensemblerrun`.

### Characteristics

Applies to	Object: SimData
Data type	struct
Data values	<p>Default values are as follows:</p> <pre> ConfigSet: [] SimulationDate: '' SimulationType: '' Variant: [] </pre> <p>In practice, the <code>ConfigSet</code>, <code>SimulationDate</code>, and <code>SimulationType</code> fields are rarely empty, since they are populated after simulation.</p>
Access	Read-only

## See Also

StopTime

## RuntimeOptions

Options for logged species

### Description

The `RuntimeOptions` property holds options for species that will be logged during the simulation run. The run-time options object can be accessed through this property.

The `LogDecimation` property of the configuration set object defines how often data is logged.

### Property Summary

`StatesToLog` Specify species, compartment, or parameter data recorded  
`Type` Display SimBiology object type

### Characteristics

Applies to	Object: <code>configset</code>
Data type	Object
Data values	Run-time options
Access	Read-only

### Examples

- 1 Create a model object, and retrieve its configuration set.

```
modelObj = sbiomodel('cell');  
configsetObj = getconfigset(modelObj);
```

- 2 Retrieve the `RuntimeOptions` object from the `configset` object.



```
runtimeObj = get(configsetObj, 'RunTimeOptions')  
Runtime Settings:
```

```
StatesToLog:      all
```

## See Also

get, set

## SensitivityAnalysis

Enable or disable sensitivity analysis

### Description

`SensitivityAnalysis` is a property of the `SolverOptions` property, which is a property of a `configset` object. This property lets you compute the time-dependent sensitivities of all the species states defined by the `StatesToLog` property with respect to the `Inputs` that you specify in the `SensitivityAnalysisOptions` property of the configuration set object.

SimBiology always uses the SUNDIALS solver to perform sensitivity analysis on a model, regardless of what you have selected as the `SolverType` in the configuration set.

---

**Note** Models containing the following active components do not support sensitivity analysis:

- Nonconstant compartments
  - Algebraic rules
  - Events
- 

For more information on setting up sensitivity analysis, see `SensitivityAnalysisOptions` . For a description of sensitivity analysis calculations, see “Sensitivity Calculation”.

### Characteristics

Applies to	Object: <code>SolverOptions</code>
Data type	logical
Data values	1, 0, true, false. Default is false.
Access	Read/write

## Examples

This example shows how to enable `SensitivityAnalysis`.

- 1 Retrieve the `configset` object from the `modelObj`.

```
modelObj = sbiomodel('cell');  
configsetObj = getConfigset(modelObj);
```

- 2 Enable `SensitivityAnalysis`.

```
set(configsetObj.SolverOptions, 'SensitivityAnalysis', true);  
get(configsetObj.SolverOptions, 'SensitivityAnalysis')
```

```
ans =
```

```
on
```

## See Also

`SensitivityAnalysisOptions`, `SolverOptions`, `SolverType`, `StatesToLog`

## SensitivityAnalysisOptions

Specify sensitivity analysis options

### Description

The `SensitivityAnalysisOptions` property is an object that holds the sensitivity analysis options in the configuration set object. Sensitivity analysis is supported only for deterministic (ODE) simulations.

---

**Note** The `SensitivityAnalysisOptions` property controls the settings related to sensitivity analysis. To enable or disable sensitivity analysis, use the `SensitivityAnalysis` property.

---

Properties of `SensitivityAnalysisOptions` are summarized in “Property Summary” on page 3-205.

When sensitivity analysis is enabled, the following command

```
[t,x,names] = sbiosimulate(modelObj)
```

returns `[t,x,names]`, where

- `t` is an  $n$ -by-1 vector, where  $n$  is the number of steps taken by the ode solver and `t` defines the time steps of the solver.
- `x` is an  $n$ -by- $m$  matrix, where  $n$  is the number of steps taken by the ode solver and  $m$  is:

Number of species and parameters specified in `StatesToLog` +  
(Number of sensitivity outputs \* Number of sensitivity input factors)

A SimBiology state includes species and nonconstant parameters.

- `names` is the list of states logged and the list of sensitivities of the species specified in `StatesToLog` with respect to the input factors.

For an example of the output, see “Examples” on page 3-205.

You can add a number of configuration set objects with different `SensitivityAnalysisOptions` to the model object with the `addconfigset` method.

Only one configuration set object in the model object can have the `Active` property set to `true` at any given time.

## Property Summary

Inputs	Specify species and parameter input factors for sensitivity analysis
Normalization	Specify normalization type for sensitivity analysis
Outputs	Specify species and parameter outputs for sensitivity analysis

## Characteristics

Applies to	Object: configuration set
Data type	Object
Data values	<code>SensitivityAnalysisOptions</code> properties as summarized in “Property Summary” on page 3-205.
Access	Read-only

## Examples

This example shows how to set `SensitivityAnalysisOptions`.

- 1 Import the radio decay model from SimBiology demos.
 

```
modelObj = sbmlimport('radiodecay');
```
- 2 Retrieve the configuration settings and the sensitivity analysis options from `modelObj`.
 

```
configsetObj = getconfigset(modelObj);
sensitivityObj = get(configsetObj, 'SensitivityAnalysisOptions');
```
- 3 Add a species and a parameter to the `Inputs` property. Use the `sbioselect` function to retrieve the species and parameter objects from the model.
- 4 Add a species to the `Outputs` property and display.
 

```
SimBiology Species Array
```

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	z	0	molecule

**5** Enable SensitivityAnalysis.

```
set(configsetObj.SolverOptions, 'SensitivityAnalysis', true);  
get(configsetObj.SolverOptions, 'SensitivityAnalysis')
```

```
ans =
```

```
1
```

**6** Simulate and return the results to three output variables. See “Description” on page 3-204 for more information.

```
[t,x,names] = sbiosimulate(modelObj);
```

**7** Display the names.

```
names
```

```
names =
```

```
 'x'
```

```
 'z'
```

```
 'd[z]/d[z]_0'
```

```
 'd[z]/d[Reaction1.c]'
```

Display state values x.

```
x
```

The display follows the column order shown in names for the values in x. The rows correspond to t.

## See Also

addconfigset, getconfigset, SensitivityAnalysis

# SolverOptions

Specify model solver options

## Description

The `SolverOptions` property is an object that holds the model solver options in the `configset` object. Changing the property `SolverType` changes the options specified in the `SolverOptions` object.

Properties of `SolverOptions` are summarized in “Property Summary” on page 3-207.

## Property Summary

<code>AbsoluteTolerance</code>	Absolute error tolerance applied to state value during simulation
<code>AbsoluteToleranceScaling</code>	Control scaling of absolute error tolerance during simulation
<code>AbsoluteToleranceStepSize</code>	Initial guess for time step size for scaling of absolute error tolerance
<code>ErrorTolerance</code>	Specify explicit or implicit tau error tolerance
<code>LogDecimation</code>	Specify frequency to log stochastic simulation output
<code>MaxIterations</code>	Specify nonlinear solver maximum iterations in implicit tau
<code>MaxStep</code>	Specify upper bound on ODE solver step size
<code>OutputTimes</code>	Specify times to log deterministic simulation output
<code>RandomState</code>	Set random number generator
<code>RelativeTolerance</code>	Allowable error tolerance relative to state value during a simulation
<code>SensitivityAnalysis</code>	Enable or disable sensitivity analysis
<code>Type</code>	Display <code>SimBiology</code> object type

## Characteristics

Applies to	Object: configset
Data type	Object
Data values	Solver options depending on SolverType. Default is SolverOptions for default SolverType (ode15s).
Access	Read-only

## Examples

This example shows the changes in SolverOptions for various SolverType settings.

- 1 Retrieve the configset object from the modelObj.

```
modelObj = sbiomodel('cell');  
configsetObj = getConfigset(modelObj);
```

- 2 Configure the SolverType to ode45.

```
set(configsetObj, 'SolverType', 'ode45');  
get(configsetObj, 'SolverOptions')
```

```
Solver Settings: (ode)
```

```
    AbsoluteTolerance:    1.000000e-006  
    RelativeTolerance:    1.000000e-003
```

- 3 Configure the SolverType to ssa.

```
set(configsetObj, 'SolverType', 'ssa');  
get(configsetObj, 'SolverOptions')
```

```
Solver Settings: (ssa)
```

```
    LogDecimation:        1  
    RandomState:          []
```

- 4 Configure the SolverType to impltau.

```
set(configsetObj, 'SolverType', 'impltau');  
get(configsetObj, 'SolverOptions')
```



Solver Settings: (impltau)

```
ErrorTolerance:    3.000000e-002
LogDecimation:    1
AbsoluteTolerance: 1.000000e-002
RelativeTolerance: 1.000000e-002
MaxIterations:    15
RandomState:      []
```

- 5 Configure the SolverType to expltau.

```
set(configsetObj, 'SolverType', 'expltau');
get(configsetObj, 'SolverOptions')
```

Solver Settings: (expltau)

```
ErrorTolerance:    3.000000e-002
LogDecimation:    1
RandomState:      []
```

## See Also

addconfigset, getconfigset

## SolverType

Select solver type for simulation

### Description

The `SolverType` property lets you specify the solver to use for a simulation. For a discussion about solver types, see “Choosing a Simulation Solver”.

Changing the solver type changes the options (properties) specified in the `SolverOptions` property of the `configset` object. If you change any `SolverOptions`, these changes are persistent when you switch `SolverType`. For example, if you set the `ErrorTolerance` for the `expltau` solver and then change to `impltau` when you switch back to `expltau`, the `ErrorTolerance` will have the value you assigned.

### Characteristics

Applies to	Object: <code>Configset</code>
Data type	enum

Data values	'ode15s', 'ode23t', 'ode45', 'sundials', 'ssa', 'expltau', 'impltau'. Default is 'ode15s'.
	<p><b>Note</b></p> <ul style="list-style-type: none"> <li>• If your model contains events, you cannot specify 'expltau' or 'impltau' for the SolverType property.</li> <li>• If your model contains doses, you cannot specify 'ssa', 'expltau', or 'impltau' for the SolverType property.</li> <li>• If your model contains algebraic rules, you cannot use 'ode45'.</li> <li>• SimBiology always uses the SUNDIALS solver to perform sensitivity analysis on a model, regardless of what you have selected as the SolverType in the configuration set.</li> </ul>
Access	Read/write

## Examples

- 1 Retrieve the configset object from the modelObj.

```
modelObj = sbiomodel('cell');
configsetObj = getconfigset(modelObj)
```

```
Configuration Settings - default (active)
  SolverType:          ode15s
  StopTime:           10.000000

  SolverOptions:
    AbsoluteTolerance: 1.000000e-006
    RelativeTolerance: 1.000000e-003
    SensitivityAnalysis: false

  RuntimeOptions:
    StatesToLog:       all

  CompileOptions:
    UnitConversion:    false
```

DimensionalAnalysis: true

SensitivityAnalysisOptions:

Inputs: 0

Outputs: 0

- 2 Configure the SolverType to ode45.

```
set(configsetObj, 'SolverType', 'ode45')
```

```
configsetObj
```

Configuration Settings - default (active)

SolverType: ode45

StopTime: 10.000000

SolverOptions:

AbsoluteTolerance: 1.000000e-006

RelativeTolerance: 1.000000e-003

SensitivityAnalysis: false

RuntimeOptions:

StatesToLog: all

CompileOptions:

UnitConversion: false

DimensionalAnalysis: true

SensitivityAnalysisOptions:

Inputs: 0

Outputs: 0

## See Also

getConfigset, set

# Species

Array of species in compartment object

## Description

The `Species` property is a property of the compartment object and indicates all the species in a compartment object. `Species` is a read-only array of SimBiology species objects.

In the model object, `Species` contains a flat list of all the species that exist within all the compartments in the model. You should always access a species through its compartment rather than the model object. Use the format `compartmentName.speciesName`, for example, `nucleus.DNA`. Another example of the syntax is `modelObj.Compartments(2).Species(1)`. The `Species` property in the model object might not be available in a future version of the software.

Species are entities that take part in reactions. A species object is added to the `Species` property when a reaction is added to the model object with the method `addreaction` on page 2-51. A species object can also be added to the `Species` property with the method `addspecies` on page 2-61.

If you remove a reaction with the method `delete` on page 2-137, and a species is no longer being used by any of the remaining reactions, the species object is *not* removed from the `Species` property. You have to use the `delete` method to remove species.

There are reserved characters that cannot be used in species object names. See `Name` for more information.

## Characteristics

Applies to	Object: compartment
Data type	Array of species objects
Data values	Species object. Default is [ ] (empty).
Access	Read-only

## **See Also**

addcompartment, addreaction, addspecies, delete

# SpeciesVariableNames

Cell array of species in reaction rate equation

## Description

The `SpeciesVariableNames` property shows the species used by the kinetic law object to determine the `ReactionRate` on page 3-177 equation in the reaction object. Use `setspecies` to assign `SpeciesVariableNames`. When you assign species to `SpeciesVariableNames`, SimBiology software maps these species names to `SpeciesVariables` on page 3-217 in the kinetic law object.

The `ReactionRate` property of a reaction object shows the result of a mapping from kinetic law definition on page 3-88. The `ReactionRate` is determined by the kinetic law object `Expression` property by mapping `ParameterVariableNames` to `ParameterVariables` and `SpeciesVariableNames` to `SpeciesVariables`.

## Characteristics

Applies to	Object: kinetic law
Data type	Cell array of character vectors
Data values	Cell array of species names
Access	Read/write

## Examples

Create a model, add a reaction, and assign the `SpeciesVariableNames` for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object, of the type 'Henri-Michaelis-Menten'

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

The reactionObj KineticLaw property is configured to kineticlawObj.

- 3 The 'Henri-Michaelis-Menten' kinetic law has one species variable (S) that you should set. To set this variable:

```
setspecies(kineticlawObj, 'S', 'a');
```

- 4 Verify that the species variable is correct.

```
get (kineticlawObj, 'SpeciesVariableNames')
```

MATLAB returns:

```
ans =
```

```
'a'
```

## See Also

Expression(AbstractKineticLaw, KineticLaw), ParameterVariables, ParameterVariableNames, ReactionRate, setparameter, SpeciesVariables



# SpeciesVariables

Species in abstract kinetic law

## Description

This property shows species variables that are used in the `Expression` on page 3-88 property of the kinetic law object to determine the `ReactionRate` on page 3-177 equation in the reaction object. Use the MATLAB function `set` to assign `SpeciesVariables` to an abstract kinetic law. For more information, see abstract kinetic law on page 3-88.

## Characteristics

Applies to	Objects: abstract kinetic law, kineticlaw
Data type	Cell array of character vectors
Data values	Defined by abstract kinetic law
Access	Read/write in abstract kinetic law. Read-only in kinetic law.

## Examples

Create a model, add a reaction, and assign the `SpeciesVariableNames` for the reaction rate equation.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
reactionObj = addreaction(modelObj, 'a -> c + d');
```

- 2 Create a kinetic law object for the reaction object, of the type 'Henri-Michaelis-Menten'.

```
kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');
```

`reactionObj` `KineticLaw` property is configured to `kineticlawObj`.

- 3 View the species variable for 'Henri-Michaelis-Menten' kinetic law.

```
get (kineticlawObj, 'SpeciesVariables')
```

MATLAB returns:

```
ans =
```

```
    'S'
```

## See Also

Expression(AbstractKineticLaw, KineticLaw), ParameterVariables,  
ParameterVariableNames, ReactionRate, set, setparameter, SpeciesVariableNames

# StartTime

Start time for initial dose time

## Description

StartTime is a property of a RepeatDose object. For a series of repeated doses, the StartTime property defines the amount of time that elapses before the first (initial) dose is given.

For RepeatDose objects, you can parameterize the property by setting it to the name of a model-scoped parameter that is not being modified by a repeated assignment rule, an algebraic rule, or a rate rule. However, the parameter can be modified by an event.

## Characteristics

Applies to	Objects: RepeatDose.
Data type	double or character vector.
Data values	Nonnegative real number or name of a model-scoped parameter object. The default value is 0.
Access	Read-write.

## See Also

RepeatDose object | ScheduleDose object

## Topics

“Parameterized and Adaptive Doses”

## StatesToLog

Specify species, compartment, or parameter data recorded

### Description

The `StatesToLog` property specifies the species, compartment, or parameter data to log during a simulation. This is the data returned in `x` during execution of `[t,x] = sbiosimulate(modelObj)`. By default, all species, nonconstant compartments, and nonconstant parameters are logged.

If you specify a particular list of species, compartments, or parameters to be logged, the order of the states in the result `SimData` after simulation is the same as the order specified.

### Characteristics

Applies to	Object: <code>RuntimeOptions</code>
Data type	Character vector, cell array of character vectors, object or vector of objects
Data values	Species objects, compartment objects, or parameter objects. Default is <code>all</code> , which means all species objects, all compartment objects whose <code>ConstantCapacity = false</code> , and all parameter objects whose <code>ConstantValue = false</code> .
Access	Read/write

### Examples

#### Specify a List of Species to be Logged During Simulation

Load the Lotka-Volterra Model.

```
sbioloadproject lotka;
```

Get the configset object of the lotka model m1.

```
configset = getconfigset(m1);
```

Display the list of species whose data are logged by default during the simulation.

```
configset.RuntimeOptions.StatesToLog
```

```
SimBiology Species Array
```

Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	x	1	
2	unnamed	y1	900	
3	unnamed	y2	900	
4	unnamed	z	0	

Suppose you want to log just species y1 and y2 data. You can specify their names as a cell array of strings and set it to StatesToLog property.

```
configset.RuntimeOptions.StatesToLog = {'y1', 'y2'};
```

Confirm the setting.

```
configset.RuntimeOptions.StatesToLog
```

```
SimBiology Species Array
```

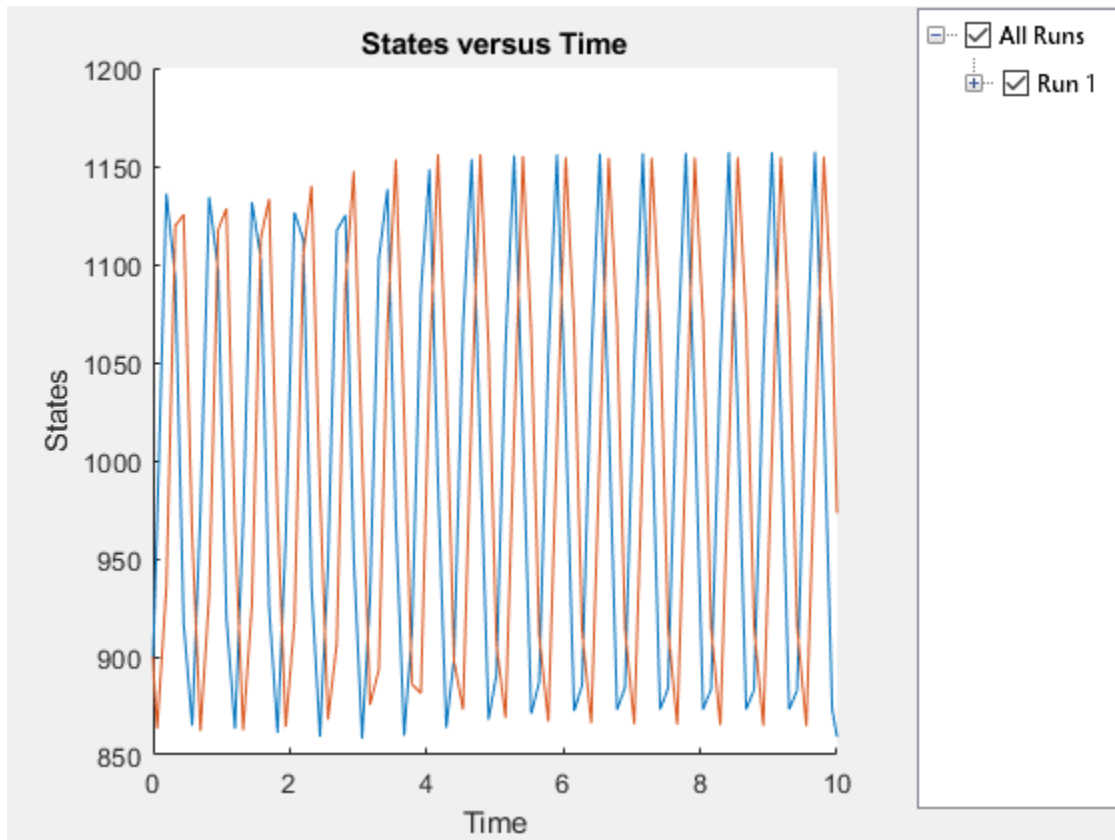
Index:	Compartment:	Name:	InitialAmount:	InitialAmountUnits:
1	unnamed	y1	900	
2	unnamed	y2	900	

Alternatively, you can specify an array of species objects (instead of strings) to StatesToLog property.

```
y1 = m1.Species(2);
y2 = m1.Species(3);
configset.RuntimeOptions.StatesToLog = [y1, y2];
```

Simulate and plot the results. Notice that simulation results of only y1 and y2 are plotted.

```
sbioplot(sbiosimulate(m1));
```

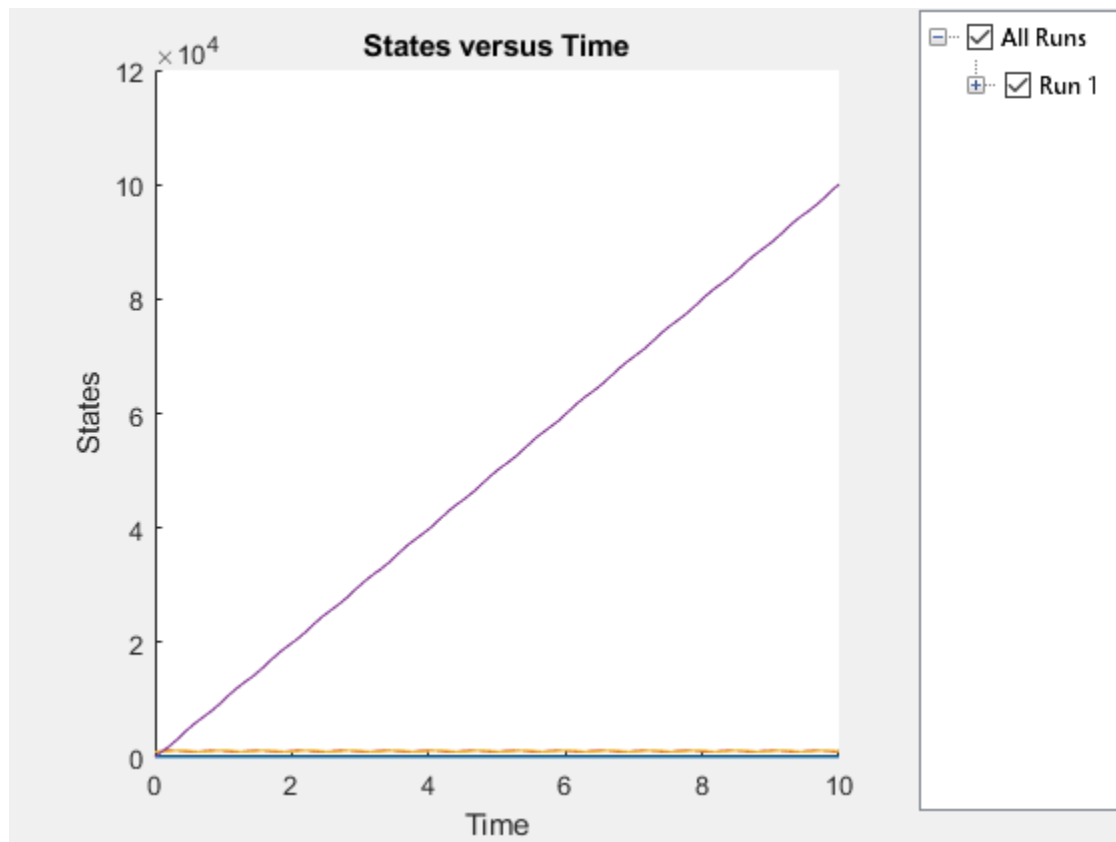


To reset to the default list, set `StatesToLog` to a string `'all'`.

```
configset.RuntimeOptions.StatesToLog = 'all';
```

Simulate again. Notice all the species data are plotted.

```
sbioplot(sbiosimulate(m1));
```



Do not specify 'all' as a cell string such as {'all'}. If so, SimBiology interprets it as a species named all.

## Stoichiometry

Species coefficients in reaction

### Description

The **Stoichiometry** property specifies the species coefficients in a reaction. Enter an array of **doubles** indicating the stoichiometry of reactants (negative value) and products (positive value). Example: [-1 -1 2].

The **double** specified cannot be 0. The reactants of the reaction are defined with a negative number. The products of the reaction are defined with a positive number. For example, the reaction  $3\text{H} + \text{A} \rightarrow 2\text{C} + \text{F}$  has the **Stoichiometry** value of [-3 -1 2 1].

When this property is configured, the **Reaction** property updates accordingly. In the above example, if the **Stoichiometry** value was set to [-2 -1 2 3], the reaction is updated to  $2\text{H} + \text{A} \rightarrow 2\text{C} + 3\text{F}$ .

The length of the **Stoichiometry** array is the sum of the **Reactants** array and the **Products** array. To remove a product or reactant from a reaction, use the **rmproduct** on page 2-424 or **rmreactant** on page 2-426 function. Add a product or reactant and set stoichiometry with methods **addproduct** on page 2-45 and **addreactant** on page 2-48.

ODE solvers support double stoichiometry values such as 0.5. Stochastic solvers and dimensional analysis currently support only integers in **Stoichiometry**, therefore you must balance the reaction equation and specify integer values for these two cases.

$\text{A} \rightarrow \text{null}$  has a stoichiometry value of [-1].  $\text{null} \rightarrow \text{B}$  has a stoichiometry value of [1].

### Characteristics

Applies to	Object: reaction
Data type	Double array



Data values	1-by-n double, where n is length (products) + length (reactants). Default is [ ] (empty).
Access	Read/write

## Examples

- 1 Create a reaction object.

```
modelObj = sbiomodel('cell');
reactionObj = addreaction(modelObj, '2 a + 3 b -> d + 2 c');
```

- 2 Verify the Reaction and Stoichiometry properties for reactionObj.

```
get(reactionObj, 'Stoichiometry')
```

MATLAB returns:

```
ans =
```

```
-2   -3    1    2
```

- 3 Set stoichiometry to [-1 -2 2 2].

```
set (reactionObj, 'Stoichiometry', [-1 -2 2 2]);
get (reactionObj, 'Stoichiometry')
```

MATLAB returns:

```
ans =
```

```
-1   -2    2    2
```

- 4 Note with get that the Reaction property updates automatically.

```
get (reactionObj, 'Reaction')
```

MATLAB returns:

```
ans =
```

```
a + 2 b -> 2 d + 2 c
```

## **See Also**

addproduct, addreactant, addreaction, Reaction, rmproduct, rmreactant

# StopTime

Simulation time criteria to stop simulation

## Description

StopTime is a property of a Configset object. This property sets the maximum simulation time criteria to stop a simulation. Time units are specified by the TimeUnits property of the Configset object.

A simulation stops when it meets any of the criteria specified by StopTime, MaximumNumberOfLogs, or MaximumWallClock. However, if you specify the OutputTimes property of the SolverOptions property of the Configset object, then StopTime and MaximumNumberOfLogs are ignored. Instead, the last value in OutputTimes is used as the StopTime criteria, and the length of OutputTimes is used as the MaximumNumberOfLogs criteria.

## Characteristics

Applies to	Object: Configset
Data type	double
Data values	Nonnegative scalar. Default is 10.
Access	Read/write

## Examples

### Set Simulation Time Criteria to Stop Simulation

- 1 Create a model object named cell and save it in a variable named modelObj. Retrieve the configuration set from modelObj and save it in a variable named configsetObj.

```
modelObj = sbiomodel('cell');
configsetObj = getconfigset(modelObj);
```

- 2 Configure the simulation stop criteria by setting the `StopTime` property to 20 seconds. Leave the `MaximumNumberOfLogs` and `MaximumWallClock` properties at their default values of `Inf`.

```
set(configsetObj, 'StopTime', 20)
get(configsetObj)
```

```
Active: 1
CompileOptions: [1x1 SimBiology.CompileOptions]
    Name: 'default'
    Notes: ''
RuntimeOptions: [1x1 SimBiology.RuntimeOptions]
SensitivityAnalysisOptions: [1x1 SimBiology.SensitivityAnalysisOptions]
SolverOptions: [1x1 SimBiology.ODESolverOptions]
    SolverType: 'ode15s'
    StopTime: 20
MaximumNumberOfLogs: Inf
MaximumWallClock: Inf
    TimeUnits: 'second'
    Type: 'configset'
```

When you simulate `modelObj`, the simulation stops when the simulation time reaches 20 seconds.

## See Also

`Configset` object, `MaximumNumberOfLogs`, `MaximumWallClock`, `OutputTimes`, `TimeUnits`, `MassUnits`, `AmountUnits`

# Tag

Specify label for SimBiology object

## Description

The `Tag` property specifies a label associated with a SimBiology object. Use this property to group objects and then use `sbiomodel` to retrieve. For example, use the `Tag` property in reaction objects to group synthesis or degradation reactions. You can then retrieve all synthesis reactions using `sbiomodel`. Similarly, for species objects you can enter and store classification information, for example, membrane protein, transcription factor, enzyme classifications, or whether a species is an independent variable. You can also enter the full form of the name of the species.

## Characteristics

Applies to	Objects: abstract kinetic law, kinetic law, model, parameter, reaction, RepeatDose, rule, ScheduleDose, species
Data type	Character vector
Data values	Any character vector
Access	Read/write

## Examples

- 1 Create a model object.
 

```
modelObj = sbiomodel ('my_model');
```
- 2 Add a reaction object and set the `Tag` property to 'Synthesis Reaction'.
 

```
reactionObj = addreaction (modelObj, 'a + b -> c + d');
set (reactionObj, 'Tag', 'Synthesis Reaction')
```
- 3 Verify the `Tag` assignment.
 

```
get (reactionObj, 'Tag');
```

MATLAB returns:

ans =

'Synthesis Reaction'

## See Also

addkineticlaw, addparameter, addreaction, addrule, addspecies, RepeatDose object, sbioabstractkineticlaw, sbiomodel, sbioroot, ScheduleDose object

## TargetName

Species receiving dose

### Description

TargetName is a property of a RepeatDose or ScheduleDose object. This property defines the SimBiology species receiving the dose. The dose amount increases the species amount at each time interval defined by a repeat dose or at each time point defined by a schedule dose.

The value of TargetName is the name of a species. If the model has more than one species with the same name, TargetName is defined as *compartmentName.speciesName*, where *compartmentName* is the name of the compartment containing the species.

### Characteristics

Applies to	Objects: RepeatDose, ScheduleDose
Data type	Character vector
Data values	Species name. Default value is "" (empty).
Access	Read/Write

### See Also

ScheduleDose object and RepeatDose object

# Trigger

Event trigger

## Description

Trigger is a property of an Event object

A **Trigger** is a condition that must become true for an event to execute. You can use a combination of relational and logical operators to build a trigger expression. **Trigger** can be a character vector, an expression, or a function handle that when evaluated returns a value of `true` or `false`. A **Trigger** can access species, parameters, and compartments.

A trigger can contain the keyword `time` and relational operators to trigger an event that occurs at a specific time during the simulation. For example, `time >= x`. In this example trigger, note that:

- The units associated with the keyword `time` are the units for the `TimeUnits` property for the `Configset` object associated with the simulation.
- If `x` is an expression containing compartments, species, or parameters, then any units associated with the expression must have the same dimensions as the keyword `time`.
- If `x` is a raw number, then its dimensions (and units, if unit conversion is on) are assumed to be the same as the keyword `time`.

For more information about how the SimBiology software handles events, see “How Events Are Evaluated”. For examples of event functions, see “Specifying Event Triggers”.

---

**Tip** If `UnitConversion` is on and your model has any event, follow the recommendation below.

Non-dimensionalize any parameters used in the event `trigger` if they are not already dimensionless. For example, suppose you have a trigger `x > 1`, where `x` is the species concentration in mole/liter. Non-dimensionalize `x` by scaling (dividing) it with a constant such as `x/x0 > 1`, where `x0` is a parameter defined as 1.0 mole/liter. Note that `x` does not have to have the same unit as the constant `x0`, but must be dimensionally consistent with it. For example, the unit of `x` can be picomole/liter instead of mole/liter.

---



## Characteristics

Applies to	Object: <code>event</code>
SimBiology type	Character vector, function handle
SimBiology values	Specify a MATLAB expression as a character vector. Default is <code>''</code> (empty character vector).
Access	Read/write

## Examples

- 1 Create a model object, and then add an event object.

```
modelObj = sbmlimport('oscillator');  
eventObj = addevent(modelObj, 'time>= 5', 'OpC = 200');
```

- 2 Set the `Trigger` property of the event object.

```
set(eventObj, 'Trigger', '(time >=5) && (speciesA<1000)');
```

- 3 Get the `Trigger` property.

```
get(eventObj, 'Trigger')
```

## See Also

Event, EventFcns

## Time

Simulation time steps or schedule dose times

### Description

Time is a property of a `SimData` or `ScheduleDose` object.

#### SimData Object

For a simulation, the `Time` property records the time steps.

#### ScheduleDose Object

For a series of scheduled doses, the `Time` property defines the times to give a dose.

A `ScheduleDose` object defines a series of doses. Each dose can have a different amount, as defined by an amount array in the `Amount` property, and given at specified times, as defined by a time array in the `Time` property. A rate array in the `Rate` property defines how fast each dose is given. At each time point in the time array, a dose is given with the corresponding amount and rate.

### Characteristics

Applies to	Objects: <code>SimData</code> , <code>ScheduleDose</code>
Data type	<code>double</code> ( <code>SimData</code> ), <code>double array</code> ( <code>ScheduleDose</code> )
Data values	Vector of doubles ( <code>SimData</code> ) Array of nonnegative real numbers. Default value is <code>[]</code> ( <code>ScheduleDose</code> )
Access	Read-only

## **See Also**

ScheduleDose object, SimData object, StopTime

## TimeUnits

Show time units for dosing and simulation

### Description

The `TimeUnits` property specifies time units for these properties:

- `StopTime` property of a `Configset` object
- `OutputTimes` and `AbsoluteToleranceStepSize` properties of the `SolverOptions` property of a `Configset` object
- `StartTime` and `Interval` properties of a `RepeatDose` object
- `Time` property of a `ScheduleDose` object
- `Time` property of a `SimData` object

---

**Note** If you change the value of the `TimeUnits` property, make sure:

- You update the values of the `Time`, `StartTime`, `Interval`, `StopTime`, and `OutputTimes` properties accordingly.
  - You update raw numbers used in any event triggers that use the keyword `time` accordingly. For more information, see `Trigger`.
  - The units, if any, associated with expressions used in any event triggers that use the keyword `time`, are consistent with the updated `TimeUnits` property. For more information, see `Trigger`.
- 

**Tip** If `UnitConversion` is on and your model has any event, follow the recommendation below.

Non-dimensionalize any parameters used in the event trigger if they are not already dimensionless. For example, suppose you have a trigger  $x > 1$ , where  $x$  is the species concentration in mole/liter. Non-dimensionalize  $x$  by scaling (dividing) it with a constant such as  $x/x_0 > 1$ , where  $x_0$  is a parameter defined as 1.0 mole/liter. Note that  $x$  does not have to have the same unit as the constant  $x_0$ , but must be dimensionally consistent with it. For example, the unit of  $x$  can be picomole/liter instead of mole/liter.

---

## Characteristics

Applies to	Objects: Configset, RepeatDose, ScheduleDose, SimData
Data type	Character vector
Data values	<p>Empty character vector or a character vector specifying any unit defined in the Units Library.</p> <p>Default value is:</p> <ul style="list-style-type: none"> <li>• <code>second</code> — properties of a Configset object or SimData object for a model object created using <code>sbiomodel</code></li> <li>• <code>hour</code> — properties of a Configset object or SimData object for a model object created from a PKModelDesign object</li> <li>• <code>' '</code> (empty character vector) — properties of RepeatDose and ScheduleDose objects</li> </ul>
Access	<p>Read/write for properties of Configset, RepeatDose, and ScheduleDose objects</p> <p>Read only for properties of SimData objects</p>

## See Also

Configset object, RepeatDose object, ScheduleDose object, SimData object, Interval, OutputTimes, StartTime, StopTime, Time, MassUnits, AmountUnits

## Type

Display SimBiology object type

## Description

The Type property indicates a SimBiology object type. When you create a SimBiology object, the value of Type is automatically defined.

For example, when a Species object is created, the value of the Type property is automatically defined as 'species'.

## Characteristics

Applies to	Objects: abstract kinetic law, compartment, configuration set, CompileOptions, event, kinetic law, model, parameter, reaction, RepeatDose, root, rule, ScheduleDose, species, RuntimeOptions, SolverOptions, unit, unitprefix, and variant
Data type	Character vector
Data values	abstract_kinetic_law, compartment, configset, compileoptions, event, kineticlaw, parameter, reaction, repeatdose, root, rule, runtimeoptions, sbiomodel, scheduledose, species, solveroptions, unit, unitprefix, and variant
Access	Read-only

## See Also

RepeatDose object, sbiomodel, sbioroot, ScheduleDose object, setactiveconfigset

# UnitConversion

Perform unit conversion

## Description

The `UnitConversion` property specifies whether to perform unit conversion for the model before simulation. It is a property of the `CompileOptions` object. `CompileOptions` holds the model's compile time options and is the object property of the `configset` object.

When `UnitConversion` is set to `true`, the SimBiology software converts the matching physical quantities to one consistent unit system in order to resolve them. This conversion is in preparation for correct simulation, but species amounts are returned in the user-specified units.

For example, consider a reaction  $a + b \rightarrow c$ . Using mass action kinetics the reaction rate is defined as  $a*b*k$  where  $k$  is the rate constant of the reaction. If you specify that initial amounts of  $a$  and  $b$  are 0.01M and 0.005M respectively, then units of  $k$  are  $1/(M*second)$ . If you specify  $k$  with another equivalent unit definition, for example,  $1/((molecules/liter)*second)$ , `UnitConversion` occurs after `DimensionalAnalysis`.

Unit conversion requires dimensional analysis. If `DimensionalAnalysis` is off, and you turn `UnitConversion` on, then `DimensionalAnalysis` is turned on automatically. If `UnitConversion` is on and you turn off `DimensionalAnalysis`, then `UnitConversion` is turned off automatically.

If `UnitConversion` fails, then you see an error when you simulate (`sbiosimulate`).

If `UnitConversion` is set to `false`, the simulation uses the given object values.

Unit conversion involving temperature supports `Celsius` as the temperature unit. Avoid using mixed temperature units as you might get an error.

## Characteristics

Applies to	Object: CompileOptions (in configset object)
Data type	boolean
Data values	true or false. Default value is false.
Access	Read/write

---

**Note** SimBiology uses units including empty units in association with DimensionalAnalysis and UnitConversion features.

- When DimensionalAnalysis and UnitConversion are both false, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.
- When DimensionalAnalysis is true and UnitConversion is false, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.
- When UnitConversion is set to true (which requires DimensionalAnalysis to be true), SimBiology performs a dimensional analysis and converts everything to consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its unit to dimensionless.

---

**Tip** If you have a custom function and UnitConversion is on, follow the recommendation below.

- Non-dimensionalize the parameters that are passed to the function if they are not already dimensionless.

Suppose you have a custom function defined as  $y = f(t)$  where  $t$  is the time in hour and  $y$  is the concentration of a species in mole/liter. When you use this function in your model to define a repeated assignment rule for instance, define it as:  $s1 = f(\text{time}/t0) * s0$ , where  $\text{time}$  is the simulation time,  $t0$  is a parameter defined as 1.0 hour,  $s0$  is a parameter defined as 1.0 mole/liter, and  $s1$  is the concentration of a species in mole/liter. Note that  $\text{time}$  and  $s1$  do not have to be in the same units as  $t0$  and  $s0$ , but they must be dimensionally consistent. For example, the  $\text{time}$  and  $s1$  units can be set to minute and picomole/liter, respectively.

---



## Examples

This example shows how to retrieve and set `unitconversion` from the default `true` to `false` in the default configuration set in a model object.

- 1 Import a model.

```
modelObj = sbmlimport('oscillator')
```

```
SimBiology Model - Oscillator
```

```
Model Components:
```

```
Models:          0
Parameters:      0
Reactions:       42
Rules:           0
Species:         23
```

- 2 Retrieve the `configset` object of the model object.

```
configsetObj = getconfigset(modelObj)
```

```
Configuration Settings - default (active)
```

```
SolverType:      ode15s
StopTime:        10.000000
```

```
SolverOptions:
```

```
AbsoluteTolerance: 1.000000e-006
RelativeTolerance: 1.000000e-003
```

```
RuntimeOptions:
```

```
StatesToLog:     all
```

```
CompileOptions:
```

```
UnitConversion:  false
DimensionalAnalysis: true
```

- 3 Retrieve the `CompileOptions` object.

```
optionsObj = get(configsetObj,'CompileOptions')
```

```
Compile Settings:
```

```
UnitConversion:  false
DimensionalAnalysis: true
```

- 4 Assign a value of false to `UnitConversion`.  

```
set(optionsObj, 'UnitConversion', true)
```

### See Also

`get`, `getConfigset`, `sbiosimulate`, `set`

## UserData

Specify data to associate with object

### Description

Property to specify data that you want to associate with a SimBiology object. The object does not use this data directly, but you can access it using the function `get` or dot notation.

### Characteristics

Applies to	Objects: abstract kinetic law, configuration set, compartment, data, event, kinetic law, model, parameter, reaction, RepeatDose, rule, ScheduleDose, species, or unit
Data type	Any
Data values	Any. Default is empty.
Access	Read/write

### See Also

RepeatDose object, `sbioabstractkineticlaw`, `sbiomodel`, `sbiroot`, `sbiunit`, `sbiunitprefix`, ScheduleDose object

## UserDefinedLibrary

Library of user-defined components

### Description

UserDefinedLibrary is a SimBiology root object property containing all user-defined components of unit, unit prefixes, and kinetic laws that you define. You can add, modify, or delete components in the user-defined library. The UserDefinedLibrary property is an object that contains the following properties:

- **Units** — Contains any user-defined units. You can specify units for compartment capacity, species amounts and parameter values, to do dimensional analysis and unit conversion during simulation. You can display the user-defined units either by using the command `sbiowhos -userdefined -unit`, or by accessing the root object.
- **UnitPrefixes** — Contains any user-defined unit prefixes. You can specify unit prefixes in combination with a valid unit for compartment capacity, species amounts and parameter values, to do dimensional analysis and unit conversion during simulation. You can display the user-defined unit prefixes either by using the command `sbiowhos -userdefined -unitprefix`, or by accessing the root object.
- **KineticLaws** — Contains any user-defined kinetic laws. Use the command `sbiowhos -userdefined -kineticlaw` to see the list of user-defined kinetic laws. You can use user-defined kinetic laws when you use the command `addkineticlaw` to create a kinetic law object for a reaction object. Refer to the kinetic law by name when you create the kinetic law object, for example, `kineticlawObj = addkineticlaw(reactionObj, 'Henri-Michaelis-Menten');`

See “Kinetic Law Definition” on page 3-88 for a definition and more information.

### Characteristics

Applies to	Object: root
Data type	object
Data values	Unit, unit prefix, and abstract kinetic law objects

Access	Read-only
--------	-----------

Characteristics for UserDefinedLibrary properties:

- Units

Applies to	UserDefinedLibrary property
Data type	Unit objects
Data values	Units
Access	Read/write

- UnitPrefixes

Applies to	UserDefinedLibrary property
Data type	Unit prefix objects
Data values	Unit prefixes
Access	Read/write

- KineticLaws

Applies to	UserDefinedLibrary property
Data type	Abstract kinetic law object
Data values	Kinetic laws
Access	Read/write

## Examples

### Example 1

This example uses the command `sbiowhos` to show the current list of user-defined components.

```
sbiowhos -userdefined -kineticlaw
sbiowhos -userdefined -unit
sbiowhos -userdefined -unitprefix
```

### Example 2

This example shows the current list of user-defined components by accessing the root object.

```
rootObj = sbioroot;  
get(rootObj.UserDefinedLibrary, 'KineticLaws')  
get(rootObj.UserDefinedLibrary, 'Units')  
get(rootObj.UserDefinedLibrary, 'UnitPrefixes')
```

### See Also

BuiltInLibrary, sbioaddtolibrary, sbioremovefromlibrary, sbioroot, sbiounit, sbiounitprefix

## Value

Assign value to parameter object

## Description

The `Value` property is the value of the parameter object. The parameter object defines an assignment that can be used by the model object and/or the kinetic law object. Create parameters and assign `Value` using the method `addparameter` on page 2-40.

## Characteristics

Applies to	Object: parameter
Data type	double
Data values	Any double. Default value is 1.0.
Access	Read/write

## Examples

Assign a parameter with a value to the model object.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel ('my_model');
```

- 2 Add a parameter to the model object (`modelObj`) with `Value` 0.5.

```
parameterObj1 = addparameter (modelObj, 'K1', 0.5)
```

MATLAB returns:

```
SimBiology Parameter Array
```

```
Index:      Name:      Value:      ValueUnits:
  1         K1         0.5
```

## **See Also**

addparameter



# ValueUnits

Parameter value units

## Description

The ValueUnits property indicates the unit definition of the parameter object Value property. ValueUnits can be one of the built-in units. To get a list of the built-in units, use the sbioshowunits on page 1-286 function. If ValueUnits changes from one unit definition to another, the Value does not automatically convert to the new units. The sbioconvertunits on page 1-29 function does this conversion.

You can add a parameter object to a model object or a kinetic law object.

## Characteristics

Applies to	Object: parameter
Data type	Character vector
Data values	Unit from units library. Default is '' (empty character vector). Note that the default value of an empty character vector means unspecified. Unspecified units are permitted during dimensional analysis, but not during unit conversion. (Use 'dimensionless' to specify dimensionless units.)
Access	Read/write

---

**Note** SimBiology uses units including empty units in association with DimensionalAnalysis and UnitConversion features.

- When DimensionalAnalysis and UnitConversion are both false, units are not used. However, SimBiology still performs a minimum level of dimensional analysis to decide whether a reaction rate is in dimensions of amount/time or concentration/time.
- When DimensionalAnalysis is true and UnitConversion is false, units (if not empty) must have consistent dimensions so that SimBiology can perform dimensional analysis. However, the units are not converted.

- When `UnitConversion` is set to `true` (which requires `DimensionalAnalysis` to be `true`), SimBiology performs a dimensional analysis and converts everything to consistent units. Hence, you must specify consistent units, and no units can be empty. If you have a dimensionless parameter, you must still set its unit to `dimensionless`.
- 

## Examples

Assign a parameter with a value to the model object.

- 1 Create a model object, and then add a reaction object.

```
modelObj = sbiomodel('my_model');
```

- 2 Add a parameter with `Value 0.5`, and assign it to the model object (`modelObj`).

```
parameterObj1 = addparameter(modelObj, 'K1', 0.5, 'ValueUnits', '1/second')
```

MATLAB returns:

SimBiology Parameter Array

Index:	Name:	Value:	ValueUnits:
1	K1	0.5	1/second

## See Also

`addparameter`, `sbioconvertunits`, `sbioshowunits`

# ZeroOrderDurationParameter

Zero-order dose absorption duration

## Description

ZeroOrderDurationParameter is a property of the PKModelMap object. It specifies the name(s) of parameter object(s) that represent the duration of absorption when the DosingType property is ZeroOrder.

Specify the name(s) of parameter object(s) that are:

- Scoped to a model
- Constant, that is, their ConstantValue property is true

When dosing multiple compartments, a one-to-one relationship must exist between the number and order of elements in the ZeroOrderDurationParameter property and the DosingType property. For a dose that is not dosed with zero-order kinetics, use '' (an empty character vector).

## Characteristics

Applies to	Object: PKModelMap
Data type	Character vector or cell array of character vectors
	<b>Tip</b> If you are not using any zero-order doses, you can set this property to a cell array of empty character vectors, or simply an empty cell array.

Data values	Name of a parameter object or empty. Default is an empty cell array.  The parameter object(s) must be: <ul style="list-style-type: none"><li>• Scoped to a model</li><li>• Constant, that is, have a <code>ConstantValue</code> property set to <code>true</code></li></ul>
Access	Read/write

## See Also

DosingType, PKModelMap object