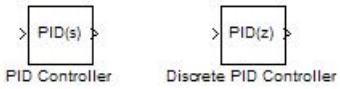
PID Controller

Simulate continuous- or discrete-time PID controllers

Library

Continuous, Discrete

Description



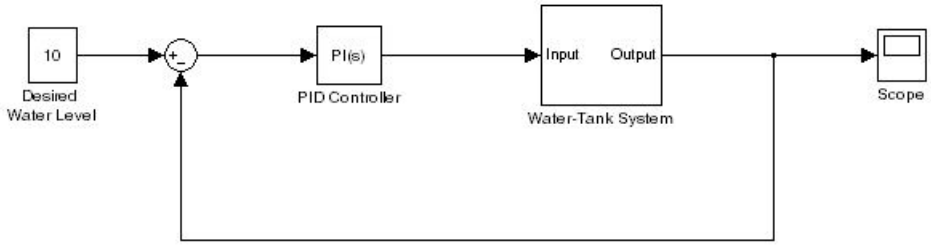
Implement a continuous- or discrete-time controller (PID, PI, PD, P, or I) in your Simulink model. PID controller gains are tunable either manually or automatically. Automatic tuning requires Simulink® Control Design™ software (PID Tuner or SISO Design Tool).

The PID Controller block output is a weighted sum of the input signal, the integral of the input signal, and the derivative of the input signal. The weights are the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action.

Configurable options in the PID Controller block include:

- Controller type and form
- Time domain (continuous or discrete)
- Initial conditions and reset trigger
- Output saturation limits and built-in anti-windup mechanism
- Signal tracking for bumpless control transfer and multiloop control

In one common implementation, the PID Controller block operates in the feedforward path of the feedback loop:



The input of the block is typically an error signal, which is the difference between a reference signal and the system output. For a two-input block that permits setpoint weighting, see the PID Controller (2 DOF) block reference page.

You can generate code to implement your controller using any Simulink data type, including fixed-point data types. (Code generation requires Real-Time Workshop software; fixed-point implementation requires Fixed-Point Toolbox.)

Data Type Support

The PID Controller block accepts real signals of any numeric data type that Simulink software supports, including fixed-point data types. See Data Types Supported by Simulink in the Simulink documentation for more information.

Parameters

The following table summarizes the PID Controller block parameters, accessible via the block parameter dialog box.

Task	Parameters
Choose controller form and type.	• **Controller Form** in Main tab • **Controller**

Task	Parameters
Choose discrete or continuous time.	• **Time-domain** • **Sample time**
Choose an integration method (discrete time).	• **Integrator method** • **Filter method**
Set and tune controller gains.	• **Proportional (P)** in Main tab • **Integral (I)** in Main tab • **Derivative (D)** in Main tab • **Filter coefficient (N)** in Main tab
Set integrator and filter initial conditions.	• **Initial conditions Source** in Main tab • **Integrator Initial condition** in Main tab • **Filter Initial condition** in Main tab • **External reset** in Main tab • **Ignore reset when linearizing** in Main tab
Limit block output.	• **Limit output** in PID Advanced tab • **Lower saturation limit** in PID Advanced tab • **Upper saturation limit** in PID Advanced tab • **Ignore saturation when linearizing** in PID Advanced tab
Configure anti-windup mechanism (when you limit block output).	• **Anti-windup method** in PID Advanced tab • **Back-calculation gain (Kb)** in PID Advanced tab
Enable signal tracking.	• **Enable tracking mode** in PID Advanced tab • **Tracking gain (Kt)** in PID Advanced tab
Configure data types.	• **Parameter data type** in Data Type Attributes tab • **Product output data type** in Data Type Attributes tab • **Summation output data type** in Data Type Attributes tab • **Accumulator data type** in Data Type Attributes tab • **Integrator output data type** in Data Type Attributes tab • **Filter output data type** in Data Type Attributes tab • **Saturation output data type** in Data Type Attributes tab • **Lock output data type setting against changes by the fixed-point tools** in Data Type Attributes tab • **Saturate on integer overflow** in Data Type Attributes tab • **Integer rounding mode** in Data Type Attributes tab

Task	Parameters
Configure block for code generation.	• State name in State Attributes tab • State name must resolve to Simulink signal object in State Attributes tab • Real-Time Workshop storage class in State Attributes tab • Real-Time Workshop storage type qualifier in State Attributes tab

Controller form

Select the controller form.

Settings

`Parallel` (Default)

Selects a controller form in which the output is the sum of the proportional, integral, and derivative actions, weighted according to the independent gain parameters **P**, **I**, and **D**. The filter coefficient **N** sets the location of the pole in the derivative filter. For a continuous-time parallel PID controller, the transfer function is:

$$C_{par}(s) = \left[P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right)\right]$$

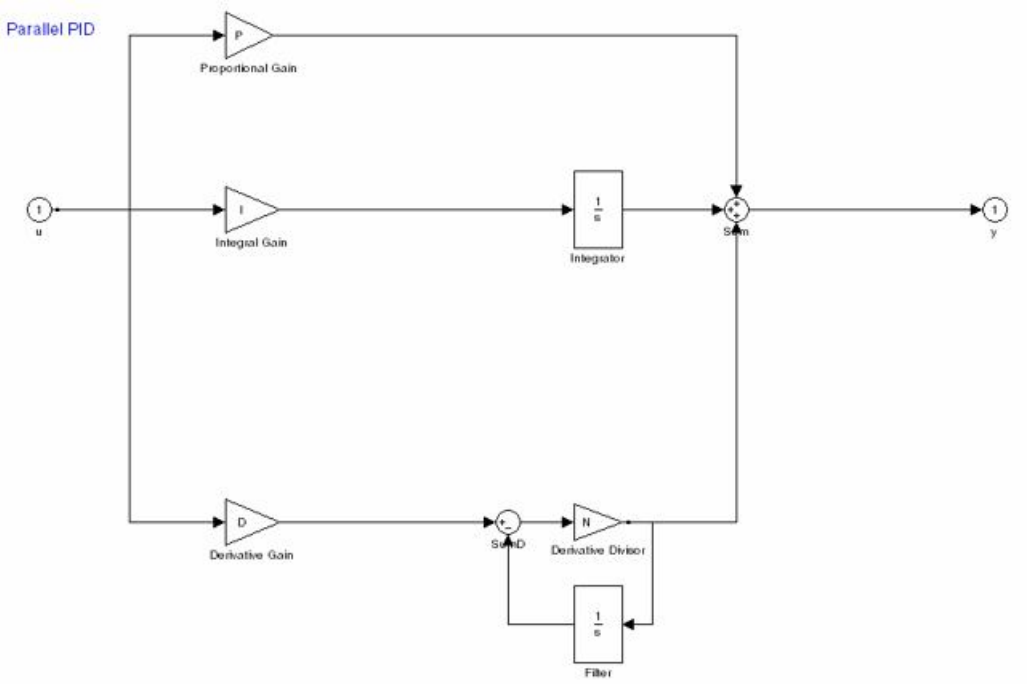
For a discrete-time parallel PID controller, the transfer function takes the form:

$$C_{par}(z) = P + Ia(z) + D\left[\frac{N}{1+Nb(z)}\right]$$

where the **Integrator method** determines `a(z)` and the **Filter method** determines `b(z)` (for sampling time T_s):

	Forward Euler method	Backward Euler method	Trapezoidal method
$a(z)$ (determined by Integrator method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2}\frac{z+1}{z-1}$
$b(z)$ (determined by Filter method)	$\frac{T_s}{z-1}$	$\frac{T_s z}{z-1}$	$\frac{T_s}{2}\frac{z+1}{z-1}$

Parallel PID Controller



`Ideal`

Selects a controller form in which the proportional gain `P` acts on the sum of all actions. The transfer functions are the same as for the parallel form, except that `P` multiplies all terms. For a continuous-time ideal PID controller, the transfer function is:

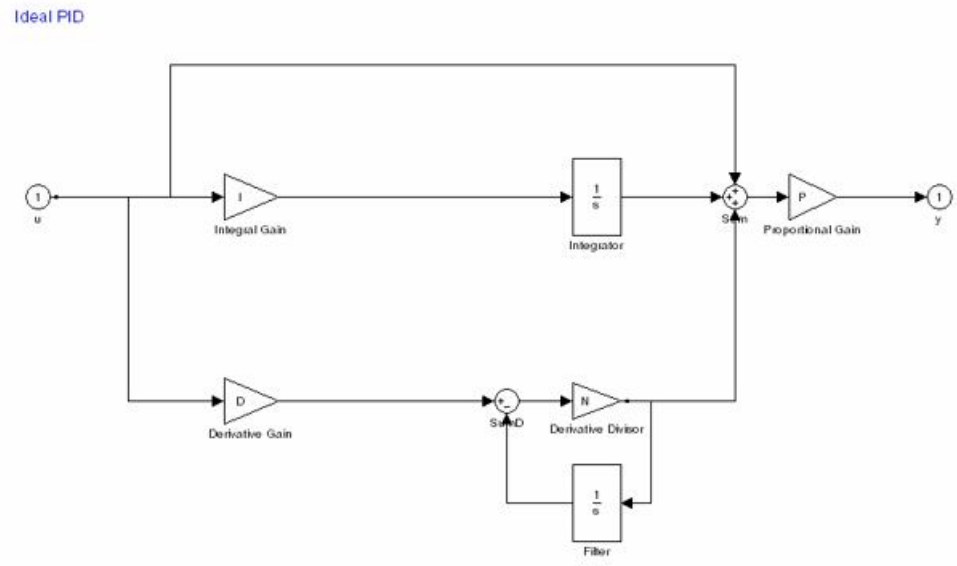
$$C_{id}(s) = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right)\right]$$

For a discrete-time ideal PID controller the transfer function is:

$$C_{id}(z) = P\left[1 + Ia(z) + D\frac{N}{1+Nb(z)}\right]$$

where the **Integrator method** determines `a(z)` and the **Filter method** determines `b(z)` as described previously for the parallel controller form.

Ideal PID Controller



Controller

Specify the controller type.

Settings

`PID` (Default)
Implements a controller with proportional, integral, and derivative action.

`PI`
Implements a controller with proportional and integral action.

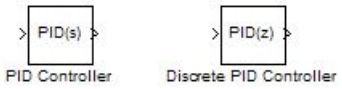
`PD`
Implements a controller with proportional and derivative action.

`P`
Implements a controller with proportional action.

`I`
Implements a controller with integral action.

Time-domain

Select continuous or discrete time domain. The appearance of the block changes to reflect your selection.



Settings

`Continuous-time` (Default)
Selects the continuous-time representation.

`Discrete-time`
Selects the discrete-time representation. Selecting `Discrete-time` also allows you to specify the:

- **Sample time**, which is the discrete interval between samples.
- Discrete integration methods for the integrator and the derivative filter using the **Integrator method** and **Filter method** menus.

Integrator method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the integrator output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Settings

`Forward Euler` (Default)
Selects the Forward Rectangular (left-hand) approximation.

- This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`
Selects the Backward Rectangular (right-hand) approximation.

- If you are generating code using Real-Time Workshop or using the Fixed-Point Toolbox and you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see Algebraic Loops in the Simulink documentation.
- An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

`Trapezoidal`
Selects the Bilinear approximation.

- If you are generating code using Real-Time Workshop or using the Fixed-Point Toolbox and you activate the `Back-calculation` **Anti-windup method**, this integration method can cause algebraic loops in your controller.

Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see Algebraic Loops in the Simulink documentation.

- An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Filter method

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the method used to compute the derivative filter output. For more information about discrete-time integration methods, see the Discrete-Time Integrator block reference page.

Settings

`Forward Euler` (Default)
Selects the Forward Rectangular (left-hand) approximation.

- This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the `Forward Euler` method can result in instability, even when discretizing a system that is stable in continuous time.

`Backward Euler`
Selects the Backward Rectangular (right-hand) approximation.

- If you are generating code using Real-Time Workshop or using the Fixed-Point Toolbox, this filter method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see Algebraic Loops in the Simulink documentation.
- An advantage of the `Backward Euler` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method.

`Trapezoidal`
Selects the Bilinear approximation.

- If you are generating code using Real-Time Workshop or using the Fixed-Point Toolbox, this filter method can cause algebraic loops in your controller. Algebraic loops can lead to slower performance of generated code. For more information about algebraic loops in Simulink models, see Algebraic Loops in the Simulink documentation.
- An advantage of the `Trapezoidal` method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Any filter parameter value `N > 0` yields a stable result with this method. Of all available filter methods, the `Trapezoidal` method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

Sample time (-1 for inherited)

(Available only when you set **Time-domain** to `Discrete-time`.) Specify the discrete interval between samples.

Settings

Default: 1

By default, the block uses a discrete sample time of 1. To specify a different sample time, enter another discrete value, such as 0.1.

If you specify a value of -1, the PID Controller block inherits the sample time from the upstream block. Do not enter a value of 0; to implement a continuous-time controller, select the **Time-domain** `Continuous-time`.

See How to Specify the Sample Time in the online documentation for more information.

Proportional (P)

(Available for PID, PD, PI, and P controllers.) Specify the proportional gain `P`.

Default: 1

Enter a finite, real gain value into the **Proportional (P)** field. Use either scalar or vector gain values. For a `Parallel` PID **Controller form**, the proportional action is independent of the integral and derivative actions. For an `Ideal` PID **Controller form**, the proportional action acts on the integral and derivative actions. See **Controller form** for more information about the role of `P` in the controller transfer function.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See Designing Compensators in the Simulink Control Design documentation.

Integral (I)

(Available for PID, PI, and I controllers.) Specify the integral gain `I`.

Default: 1

Enter a finite, real gain value into the **Integral (I)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See Designing Compensators in the Simulink Control Design documentation.

Derivative (D)

(Available for PID and PD controllers.) Specify the derivative gain `D`.

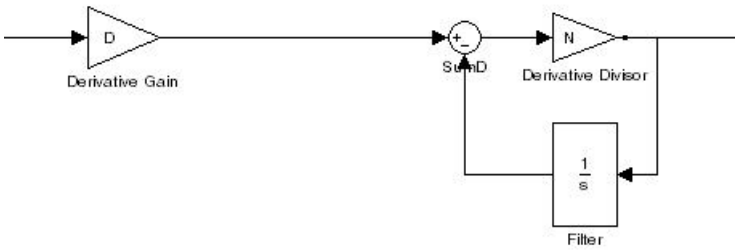
Default: 0

Enter a finite, real gain value into the **Derivative (D)** field. Use either scalar or vector gain values.

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See Designing Compensators in the Simulink Control Design documentation.

Filter coefficient (N)

(Available for PID and PD controllers.) Specify the filter coefficient `N`, which determines the pole location of the filter in the derivative action:



The filter pole falls at `s = -N` in the `Continuous-time` **Time-domain**. For `Discrete-time`, the location of the pole depends on which **Filter method** you select (for sampling time T_s):

- `Forward Euler`:
  $z_{pole} = 1 - NT_s$
- `Backward Euler`:
  $z_{pole} = \frac{1}{1 + NT_s}$
- `Trapezoidal`:
  $z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2}$

Default: 100.

Enter a finite, real gain value into the **Filter Coefficient (N)** field. Use either scalar or vector gain values. Note that the PID controller block does not support `N = inf` (ideal unfiltered derivative).

When you have Simulink Control Design software installed, you can automatically tune the controller gains using the PID Tuner or the SISO Design Tool. See Designing Compensators in the Simulink Control Design documentation. Automatic tuning requires `N > 0`.

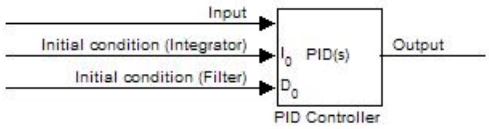
Initial conditions Source

(Only available for controllers with integral or derivative action.) Select the source of the integrator and filter initial conditions. Simulink uses initial conditions to initialize the integrator and filter output at the start of a simulation or at a specified trigger event (See **External Reset**). The integrator and filter initial conditions in turn determine the initial block output.

Settings

`internal` (Default)
Specifies the integrator and filter initial conditions explicitly using the **Integrator Initial condition** and **Filter Initial condition** parameters.

`external`
Specifies the integrator and filter initial conditions externally. An additional input port appears under the block input for each initial condition: I_0 for the integrator and D_0 for the filter:



Integrator Initial condition

(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the integrator initial value. Simulink uses the initial condition to initialize the integrator output at the start of a simulation or at a specified trigger event (see **External Reset**). The integrator initial condition, together with the filter initial condition, determines the initial output of the PID controller block.

Default: 0

Simulink does not permit the integrator initial condition to be `inf` or `NaN`.

Filter Initial condition

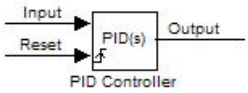
(Available only when **Initial conditions Source** is `internal` and the controller includes integral action.) Specify the filter initial value. Simulink uses the initial condition to initialize the filter output at the start of a simulation or at a specified trigger event (see **External Reset**). The filter initial condition, together with the integrator initial condition, determines the initial output of the PID controller block.

Default: 0

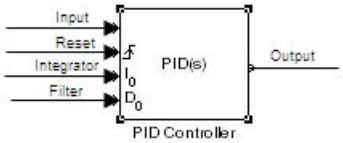
Simulink does not permit the filter initial condition to be `inf` or `NaN`.

External reset

Select the trigger event that resets the integrator and filter outputs to the initial conditions you specify in the **Integrator Initial condition** and **Filter Initial condition** fields. Selecting any option other than `none` enables a reset input on the block for the external reset signal, as shown:



Or, if the **Initial conditions Source** is `External`,



Settings

`none` (Default)
Does not reset the integrator and filter outputs to initial conditions.

`rising`
Resets the outputs when the reset signal has a rising edge.

`falling`
Resets the outputs when the reset signal has a falling edge.

`either`
Resets the outputs when the reset signal either rises or falls.

`level`
Resets and holds the outputs to the initial conditions while the reset signal is nonzero.

> **Note** To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller block.

Ignore reset when linearizing

Force Simulink linearization commands to ignore any reset mechanism that you have chosen with the **External reset** menu. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the PID Controller block to reset.

Settings

☐ `Off` (Default)
Simulink linearization commands do not ignore states corresponding to the reset mechanism.

☑ `On`
Simulink linearization commands ignore states corresponding to the reset mechanism.

Enable zero-crossing detection

Enable zero-crossing detection in continuous-time models upon reset and upon entering or leaving a saturation state.

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate an **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. For more information, see Zero-Crossing Detection.

Settings

☑ `On` (Default)
Uses zero-crossing detection at any of the following events: reset; entering or leaving an upper saturation state; and entering or leaving a lower saturation state.

☐ `Off`
Does not use zero-crossing detection.

Enabling zero-crossing detection for the PID Controller block also enables zero-crossing detection for all under-mask blocks that include the zero-crossing detection feature.

Limit output

Limit the block output to values you specify as the **Lower saturation limit** and **Upper saturation limit** parameters.

Activating this option limits the block output internally to the block, obviating the need for a separate Saturation block after the controller in your Simulink model. It also allows you to activate the block's built-in anti-windup mechanism (see **Anti-windup method**).

Settings

☐ `Off` (Default)
Does not limit the block output, which equals the weighted sum of the proportional, integral, and derivative actions.

☑ `On`
Limits the block output to the **Lower saturation limit** or the **Upper saturation limit** whenever the weighted sum exceeds those limits. Allows you to select an **Anti-windup method**.

Lower saturation limit

(Available only when you select the **Limit Output** box.) Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that

value.

Default: `-inf`

Upper saturation limit

(Available only when you select the **Limit Output** box.) Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

Default: `inf`

Anti-windup method

(Available only when you select the **Limit Output** option and the controller includes integral action.) Select an anti-windup mechanism to discharge the integrator when the block is saturated, which occurs when the sum of the block components exceeds the output limits.

When you select the **Limit output** check box and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator wind-up), increasing the difference between the block output and the sum of the block components. Without a mechanism to prevent integrator wind-up, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to discharge the integrator and return the weighted sum within the block saturation limit.

In both cases, controller performance can suffer. To combat the effects of wind-up without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. Activating an anti-windup mechanism can improve controller performance.

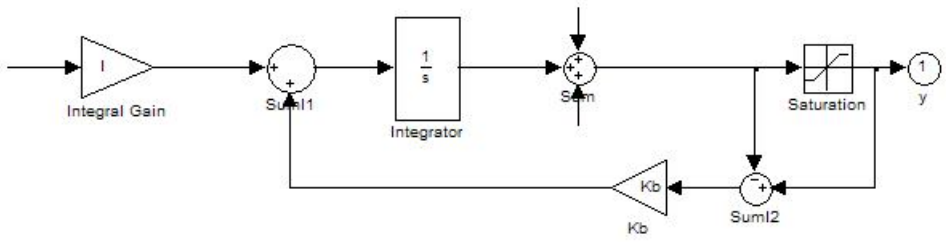
Settings

`none` (Default)

Does not use an anti-windup mechanism. This setting may cause the block's internal signals to be unbounded even if the output appears to be bounded by the saturation limits. This can result in slow recovery from saturation or unexpected overflows.

`back-calculation`

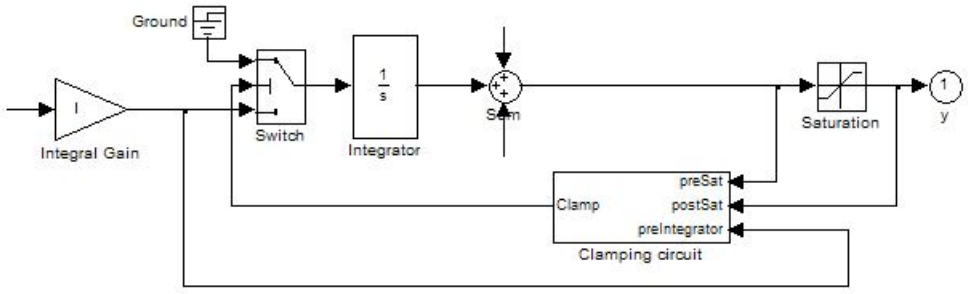
Discharges the integrator when the block output saturates using this feedback loop:



You can also specify a value for the **Back-calculation coefficient (Kb)**.

`clamping`

Stops integration when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Resumes integration when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. The integrator portion of the block is:



where the clamping circuit implements the logic necessary to determine whether integration continues.

Back-calculation gain (Kb)

(Available only when the `back-calculation` **Anti-windup method** is active.) Specify the gain coefficient of the anti-windup feedback loop.

The `back-calculation` anti-windup method discharges the integrator on block saturation using a feedback loop having gain coefficient `Kb`.

Default: 1

Ignore saturation when linearizing

Force Simulink linearization commands ignore PID Controller block output limits. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the PID Controller block to exceed the output limits.

Settings

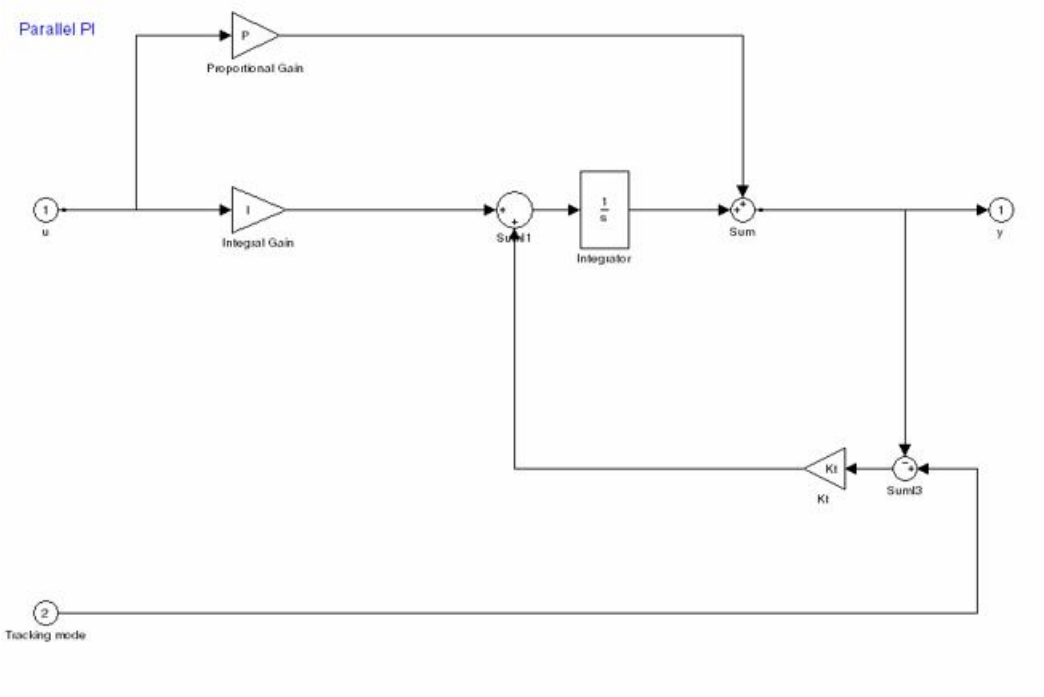
☐ `Off` (Default)
Simulink linearization commands do not ignore states corresponding to saturation.

☑ `On`
Simulink linearization commands ignore states corresponding to saturation.

Enable tracking mode

(Available for any controller with integral action.) Activate signal tracking, which lets the output of the PID Controller block follow a tracking signal. Provide the tracking signal to the block at the `TR` port, which becomes active when you select **Enable tracking mode**.

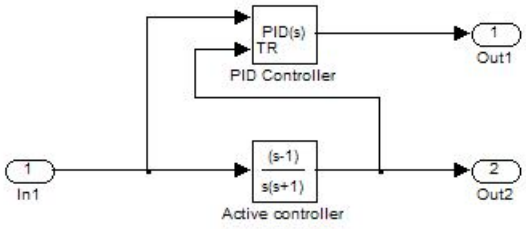
When signal tracking is active, the difference between the tracked signal and the block output is fed back to the integrator input with a gain `Kt`. The structure is illustrated for a PI controller:



You can also specify the **Tracking coefficient (Kt)**.

Bumpless control transfer

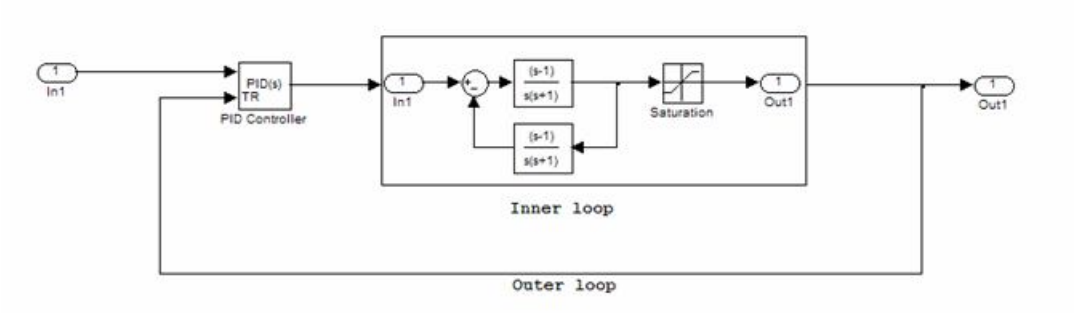
Use signal tracking, for example, to achieve bumpless control transfer in systems that switch between two controllers. You can make one controller track the output of the other controller by connecting `TR` port to the signal to be tracked. For example:



In this example, the outputs `Out1` and `Out2` can drive a controlled system (not shown) through a switch that transfers control between the "Active controller" block and the PID Controller block. The signal tracking feature of the PID Controller block provides smooth operation upon transfer of control from one controller to another, ensuring that the two controllers have the same output at the time of transfer.

Multiloop control

Use signal tracking to prevent block wind-up in multiloop control approaches, as this example illustrates:



In this example, inner loop has an effective gain of 1 when it is not saturated. Without signal tracking, inner loop winds up in saturation. Signal tracking ensures that the PID Controller output does not exceed the saturated output of the inner loop.

Settings

`Off` (Default)
Disables signal tracking and removes `TR` block input.

`On`
Enables signal tracking and activates `TR` input.

Tracking gain (Kt)

(Available only when you select **Enable tracking mode**.) Specify `Kt`, which is the gain of the signal tracking feedback loop.

Default: 1

Parameter data type

Select the data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt**.

See Data Types Supported by Simulink in the Simulink documentation for more information.

Settings

`Inherit: Inherit via internal rule` (Default)
Simulink software chooses a combination of output scaling and data type that requires the smallest amount of memory. This memory requirement accommodates the calculated output range and maintains the output precision of the block and word size of the targeted hardware implementation specified for the model. If the **Device type** parameter on the **Hardware Implementation** configuration parameters pane is set to `ASIC/FPGA`, Simulink software chooses the output data type without regard to hardware constraints. Otherwise, Simulink software chooses the smallest available hardware data type capable of meeting the range and precision constraints. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)` (a generic 32-bit microprocessor) is the specified target hardware, the output data type is `int32`.

`Inherit: Inherit via back propagation`
Use data type of the driving block.

`Inherit: Same as input`
Use data type of input signal.

```
double
single
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16)
fixdt(1,16,0)
fixdt(1,16,2^0,0)
<data type expression>
```
Name of a data type object. For example, `Simulink.NumericType`.

Product output data type

Select the product output data type of the gain parameters **P**, **I**, **D**, **N**, **Kb**, and **Kt** .

See Data Types Supported by Simulink in the Simulink documentation for more information.

Settings

`Inherit: Inherit via internal rule` (Default)
Simulink software chooses a combination of output scaling and data type that requires the smallest amount of memory. This memory requirement accommodates the calculated output range and maintains the output precision of the block and word size of the targeted hardware implementation specified for the model. If the **Device type** parameter on the **Hardware Implementation** configuration parameters pane is set to `ASIC/FPGA`, Simulink software chooses the output data type without regard to hardware constraints. Otherwise, Simulink software chooses the smallest available hardware data type capable of meeting the range and precision constraints. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If

`Unspecified (assume 32-bit Generic)` (a generic 32-bit microprocessor) is the specified target hardware, the output data type is `int32`.

`Inherit: Inherit via back propagation`
Use data type of the driving block.

`Inherit: Same as input`
Use data type of input signal.

`double`
`single`
`int8`
`uint8`
`int16`
`uint16`
`int32`
`uint32`
`fixdt(1,16)`
`fixdt(1,16,0)`
`fixdt(1,16,2^0,0)`
`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Summation output data type

Select the summation output data type of the sums **Sum**, **Sum D**, **Sum I1** , **SumI2** ,and **SumI3**, which are sums computed internally within the block. To see where Simulink computes each of these sums , right-click the PID Controller block in your model and select `Look Under Mask`:

- **Sum** is the weighted sum of the proportional, derivative, and integral signals.
- **SumD** is the sum in the derivative filter feedback loop.
- **SumI1** is the sum of the block input signal (weighted by the integral gain `I`) and **SumI2**. **SumI1** is computed only when **Limit output** and **Anti-windup method** `back-calculation` are active.
- **SumI2** is the difference between the weighted sum **Sum** and the limited block output. **SumI2** is computed only when **Limit output** and **Anti-windup method** `back-calculation` are active.
- **SumI3** is the difference between the block output and the signal at the block's tracking input. **SumI3** is computed only when you select the **Enable tracking mode** box.

See Data Types Supported by Simulink in the Simulink documentation for more information.

Settings

`Inherit: Inherit via internal rule` (Default)
Simulink software chooses a combination of output scaling and data type that requires the smallest amount of memory. This memory requirement accommodates the calculated output range and maintains the output precision of the block and word size of the targeted hardware implementation specified for the model. If the **Device type** parameter on the **Hardware Implementation** configuration parameters pane is set to `ASIC/FPGA`, Simulink software chooses the output data type without regard to hardware constraints. Otherwise, Simulink software chooses the smallest available hardware data type capable of meeting the range and precision constraints. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)` (a generic 32-bit microprocessor) is the specified target hardware, the output data type is `int32`.

`Inherit: Same as first input`
Use data type of first input signal.

`double`
`single`
`int8`
`uint8`
`int16`
`uint16`
`int32`
`uint32`

`fixdt(1,16)`
`fixdt(1,16,0)`
`fixdt(1,16,2^0,0)`
`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Accumulator data type

Specify the accumulator data type.

Settings

Default: `Inherit: Inherit via internal rule`

`Inherit: Inherit via internal rule`
Use internal rule to determine accumulator data type.

`Inherit: Same as first input`
Use data type of first input signal.

`double`
Accumulator data type is `double`.

`single`
Accumulator data type is `single`.

`int8`
Accumulator data type is `int8`.

`uint8`
Accumulator data type is `uint8`.

`int16`
Accumulator data type is `int16`.

`uint16`
Accumulator data type is `uint16`.

`int32`
Accumulator data type is `int32`.

`uint32`
Accumulator data type is `uint32`.

`fixdt(1,16,0)`
Accumulator data type is fixed point `fixdt(1,16,0)`.

`fixdt(1,16,2^0,0)`
Accumulator data type is fixed point `fixdt(1,16,2^0,0)`.

`<data type expression>`
The name of a data type object, for example `Simulink.NumericType`

Command-Line Information

See Block-Specific Parameters for the command-line information.

See Also

See Using the Data Type Assistant in the *Simulink User's Guide* for more information.

Integrator output data type

Select the data type of the integrator output.

See Data Types Supported by Simulink in the Simulink documentation for more information.

Settings

`Inherit: Inherit via internal rule` (Default)

Simulink software chooses a combination of output scaling and data type that requires the smallest amount of memory. This memory requirement accommodates the calculated output range and maintains the output precision of the block and word size of the targeted hardware implementation specified for the model. If the **Device type** parameter on the **Hardware Implementation** configuration parameters pane is set to `ASIC/FPGA`, Simulink software chooses the output data type without regard to hardware constraints. Otherwise, Simulink software chooses the smallest available hardware data type capable of meeting the range and precision constraints. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)` (a generic 32-bit microprocessor) is the specified target hardware, the output data type is `int32`.

`Inherit: Same as input`
Use data type of input signal.

`double`
`single`
`int8`
`uint8`
`int16`
`uint16`
`int32`
`uint32`
`fixdt(1,16)`
`fixdt(1,16,0)`
`fixdt(1,16,2^0,0)`
`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Filter output data type

Select the data type of the filter output.

See Data Types Supported by Simulink in the Simulink documentation for more information.

Settings

`Inherit: Inherit via internal rule` (Default)
Simulink software chooses a combination of output scaling and data type that requires the smallest amount of memory. This memory requirement accommodates the calculated output range and maintains the output precision of the block and word size of the targeted hardware implementation specified for the model. If the **Device type** parameter on the **Hardware Implementation** configuration parameters pane is set to `ASIC/FPGA`, Simulink software chooses the output data type without regard to hardware constraints. Otherwise, Simulink software chooses the smallest available hardware data type capable of meeting the range and precision constraints. For example, if the block multiplies an input of type `int8` by a gain of `int16` and `ASIC/FPGA` is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)` (a generic 32-bit microprocessor) is the specified target hardware, the output data type is `int32`.

`Inherit: Same as input`
Use data type of input signal.

`double`
`single`
`int8`
`uint8`
`int16`
`uint16`
`int32`
`uint32`
`fixdt(1,16)`
`fixdt(1,16,0)`
`fixdt(1,16,2^0,0)`
`<data type expression>`
Name of a data type object. For example, `Simulink.NumericType`.

Saturation output data type

Select the saturation output data type.

See Data Types Supported by Simulink in the Simulink documentation for more information.

Settings

`Inherit: Same as input` (Default)
Use data type of input signal.

```
double
single
int8
uint8
int16
uint16
int32
uint32
fixdt(1,16)
fixdt(1,16,0)
fixdt(1,16,2^0,0)
<data type expression>
```
Name of a data type object. For example, `Simulink.NumericType`.

Lock output data type setting against changes by the fixed-point tools

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor.

Settings

Default: Off

☑ On
Locks the output data type setting for this block.

☐ Off
Allows the Fixed-Point Tool and the Fixed-Point Advisor to change the output data type setting for this block.

Command-Line Information

See Block-Specific Parameters for the command-line information.

See Also

For more information, see Fixed-Point Tool and Fixed-Point Advisor in the Simulink Fixed Point documentation.

Saturate on integer overflow

Specify whether overflows saturate.

Settings

Default: Off

☑ On
Overflows saturate.

☐ Off
Overflows do not saturate.

Tips

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible, in which case, no saturation code is necessary.

Command-Line Information

See Block-Specific Parameters for the command-line information.

Integer rounding mode

Select the rounding mode for fixed-point operations.

Settings

Default: `Floor`

`Ceiling`
Round both positive and negative numbers toward positive infinity.

`Convergent`
Round number to the nearest representable value. If a tie occurs, round to the nearest even stored value.

`Floor`
Round both positive and negative numbers toward negative infinity.

`Nearest`
Round number to the nearest representable value. If a tie occurs, round toward positive infinity.

`Round`
Round number to the nearest representable value. If a tie occurs, round positive numbers toward positive infinity and round negative numbers toward negative infinity.

`Simplest`
This option provides for an optimization of the rounding code for several blocks.

`Zero`
Round number toward zero.

Command-Line Information

See Block-Specific Parameters for the command-line information.

See Also

For more information, see Rounding in the *Simulink Fixed Point User's Guide*.

State name

Assign unique name to each state. The state names apply only to the selected block.

To assign a name to a single state, enter the name between quotes; for example, `'velocity'`.

To assign names to multiple states, enter a comma-delimited list surrounded by braces; for example, `{'a', 'b', 'c'}`. Each name must be unique. To assign state names with a variable that has been defined in the MATLAB workspace, enter the variable without quotes. The variable can be a string, cell, or structure.

Settings

Default: `' '` (no name)

State name must resolve to Simulink signal object

Require that state name resolve to Simulink signal object.

Settings

Default: Off

☑ On
Require that state name resolve to Simulink signal object.

☐ Off
Do not require that state name resolve to Simulink signal object.

Dependencies

State name enables this parameter.

This parameter enables **Real-Time Workshop storage class**.

Command-Line Information

See Block-Specific Parameters for the command-line information.

Real-Time Workshop storage class

Select state storage class.

Settings

Default: `Auto`

`Auto`
Auto is the appropriate storage class for states that you do not need to interface to external code.

`ExportedGlobal`
State is stored in a global variable

`ImportedExtern`
`model``_private.h` declares the state as an extern variable.

`ImportedExternPointer`
`model``_private.h` declares the state as an extern pointer.

Dependencies

State name enables this parameter.

Setting this parameter to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables **Real-Time Workshop storage type qualifier**.

Command-Line Information

See Block-Specific Parameters for the command-line information.

See Also

Block State Storage Classes in the *Real-Time Workshop User's Guide*.

Real-Time Workshop storage type qualifier

Specify Real-Time storage type qualifier.

Settings

Default: `' '`

If left blank, no qualifier is assigned.

Dependencies

Setting **Real-Time Workshop storage class** to `ExportedGlobal`, `ImportedExtern`, or `ImportedExternPointer` enables this parameter.

Command-Line Information

See Block-Specific Parameters for the command-line information.

Characteristics

Direct Feedthrough	The following ports support direct feedthrough: • Reset port • Integrator and filter initial condition port • Input port, for every integration method except Forward Euler
Sample Time	Specified in the **Sample time** parameter
Scalar Expansion	Supported for gain parameters **P**, **I**, and **D** and for filter coefficient **N**
States	Inherited from driving block and parameters

Dimensionalized	Yes
Zero-Crossing Detection	Yes (in continuous-time domain)

See Also

PID Controller (2 DOF), Gain, Integrator, Discrete-Time Integrator, Derivative, Discrete Derivative.

Provide feedback about this page

Permute Dimensions PID Controller (2 DOF)