最新のリリースでは、このページがまだ翻訳されていません。 このページの最新版は英語でご覧になれます。

Discrete-Time Integrator

信号の離散時間積分または累積の実行

ライブラリ: Simulink / Commonly Used Blocks
Simulink / Discrete
HDL Coder / Discrete
HDL Coder / HDL Floating Point Operations

説明

Discrete-Time Integrator ブロックを Integrator ブロックの代わりに使用して、純粋な離散モデルを作成します。Discrete-Time Integrator ブロックを使用すると、以下のことが可能です。

- 初期条件をブロック ダイアログ ボックスまたはブロックへの入力で定義する
- 入力ゲイン (K) の値を定義する
- ブロックの状態を出力する
- 積分の上限と下限を定義します。
- 追加リセット入力を使用した状態のリセット

出力方程式

最初のタイム ステップでは、ブロックの状態は `n = 0` であり、**[初期条件設定]** パラメーターの値に応じて、初期出力 `y(0) = IC` または初期状態 `x(0) = IC` のどちらかです。

シミュレーション時間 `t(n)` のステップ `n > 0` では、Simulink® は出力 `y(n)` を以下のように更新します。

- 前進オイラー法:

```
y(n) = y(n-1) + K*[t(n) - t(n-1)]*u(n-1)
```

- 後退オイラー法:

```
y(n) = y(n-1) + K*[t(n) - t(n-1)]*u(n)
```

- 台形法:

```
y(n) = y(n-1) + K*[t(n)-t(n-1)]*[u(n)+u(n-1)]/2
```

Simulink はブロックの明示的なサンプル時間またはトリガーのサンプル時間に応じて、これらの出力方程式の状態空間表現を自動的に選択します。明示的なサンプル時間を使用する場合、`n > 0` のとき、`t(n)-t(n-1)` はサンプル時間 `T` に減少します。

積分および累積方法

このブロックは、前進オイラー法、後退オイラー法または台形則を使用して積分または累積を行うことができます。ここで、`u` は入力、`y` は出力、`x` は状態です。Simulink は、与えられたステップ `n` に対して、`y(n)` と `x(n+1)` を更新します。積分モードでは、`T` はブロックのサンプル時間 (トリガー付きサンプル時間の場合は `T` の差分) です。累積モードでは `T = 1` です。ブロックのサンプル時間は出力が計算されるタイミングを決めますが、出力の値は決めません。`K` はゲイン値です。出力値は上限値または下限値に従って制限されます。

前進オイラー法

前進オイラー法 (既定の設定) は、前進矩形近似または左側近似とも呼ばれます。

`1/s` は `T/(z-1)` で近似されます。ステップ `n` でのブロックの出力の式は次のようになります。

```
x(n+1) = x(n) + K*T*u(n)
y(n)   = x(n)
```

ブロックは以下のステップで出力を計算します。

```
Step 0:          y(0)   = IC (clip if necessary)
                 x(1)   = y(0) + K*T*u(0)

Step 1:          y(1)   = x(1)
                 x(2)   = x(1) + K*T*u(1)

Step n:          y(n)   = x(n)
                 x(n+1) = x(n) + K*T*u(n) (clip if necessary)
```

この手法では、入力端子 1 には直達がありません。

後退オイラー法

後退オイラー法は、後退矩形近似または右側近似とも呼ばれます。

`1/s` は `T*z/(z-1)` で近似されます。これにより、ステップ `n` でのブロックの出力の式は、次のようになります。

```
y(n) = y(n-1) + K*T*u(n).
```

`x(n) = y((n)-1)` とします。ブロックは以下のステップで出力を計算します。

- Triggered Subsystem と Function-call Subsystem のパラメーター **[初期条件設定]** が [出力] または [自動] に設定されている場合。

  ```
  Step 0:          y(0) = IC (clipped if necessary)
                   x(1) = y(0)
  ```

- Non-Triggered Subsystem のパラメーター **[初期条件設定]** が [自動] に設定されている場合。

  ```
  Step 0:          x(0)   = IC (clipped if necessary)
                   x(1)   = y(0) = x(0) + K*T*u(0)

  Step 1:          y(1)   = x(1) + K*T*u(1)
                   x(2)   = y(1)

  Step n:          y(n)   = x(n) + K*T*u(n)
                   x(n+1) = y(n)
  ```

この手法では、入力端子 1 には直達があります。

台形則

この手法では、`1/s` は `T/2*(z+1)/(z-1)` で近似されます。

`T` が固定の場合 (サンプリング周期に等しい場合)、出力の計算式は次のようになります。

```
x(n) = y(n-1) + K*T/2*u(n-1)
y(n) = x(n) + K*T/2*u(n)
```

- Triggered Subsystem と Function-call Subsystem のパラメーター **[初期条件設定]** が [出力] または [自動] に設定されている場合。

  ```
  Step 0:          y(0)   = IC (clipped if necessary)
                   x(1)   = y(0) + K*T/2*u(0)
  ```

- Non-Triggered Subsystem のパラメーター **[初期条件設定]** が [自動] に設定されている場合。

```
Step 0:          x(0)   = IC (clipped if necessary)
                 y(0)   = x(0) + K*T/2*u(0)
                 x(1)   = y(0) + K*T/2*u(0)

Step 1:          y(1)   = x(1) + K*T/2*u(1)
                 x(2)   = y(1) + K*T/2*u(1)

Step n:          y(n)   = x(n) + K*T/2*u(n)
                 x(n+1) = y(n) + K*T/2*u(n)
```

ここで `x(n+1)` は次の出力値の最良の推定値です。`x(n)` と `y(n)` は等しくないことから、状態と同じではありません。

この手法では、入力端子 1 には直達があります。

T が可変の場合

T が可変の場合 (たとえばトリガー時間から得られる場合)、ブロックは次のステップを使用して出力を計算します。

- Triggered Subsystem と Function-call Subsystem のパラメーター **[初期条件設定]** が [出力] または [自動] に設定されている場合。

  ```
  Step 0:          y(0)   = IC (clipped if necessary)
                   x(1)   = y(0)
  ```

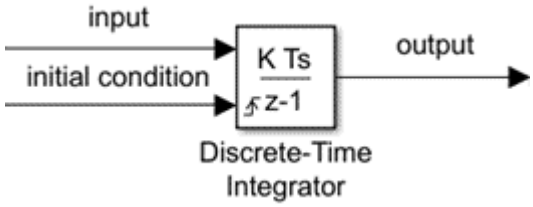
- Non-Triggered Subsystem のパラメーター **[初期条件設定]** が [自動] に設定されている場合。

  ```
  Step 0:          x(0)   = IC (clipped if necessary)
                   x(1)   = y(0) = x(0) + K*T/2*u(0)

  Step 1:          y(1)   = x(1) + T/2*(u(1) + u(0))
                   x(2)   = y(1)

  Step n:          y(n)   = x(n) + T/2*(u(n) + u(n-1))
                   x(n+1) = y(n)
  ```

初期条件の定義

初期条件は、このブロックのダイアログ ボックスでパラメーターとして定義するか、または外部信号から入力できます。

- ブロック パラメーターとして初期条件を定義するには、**[初期条件のソース]** パラメーターを [内部] と指定し、値を **[初期条件]** テキスト ボックスに入力します。
- 外部ソースから初期条件を与えるには、**[初期条件のソース]** パラメーターを [外部] に設定します。ブロックに追加の入力端子が表示されます。

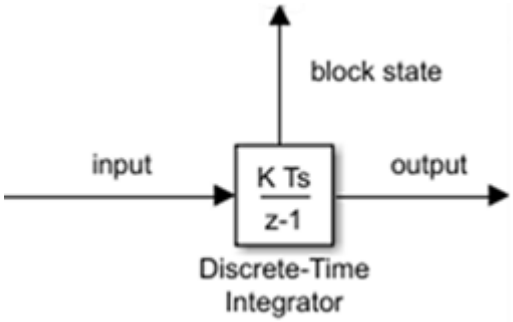


状態端子を使用する場合

出力端子ではなく状態端子を使用します。

- ブロックの出力がリセット端子または初期条件端子を通りブロックにフィードバックされ、代数ループが生じる場合。例については、`sldemo_bounce_two_integrators` モデルを参照してください。
- 条件付き実行サブシステムの 1 つから他のサブシステムに状態を渡したい場合。この場合、タイミングの問題が生じることがあります。例については、`sldemo_clutch` モデルを参照してください。

これらの問題は、出力端子ではなく状態端子を通して状態を渡すことによって解決できます。Simulink は、出力とはわずかに異なる時間に状態を生成し、これらの問題からモデルを保護します。ブロックの状態を出力するには、**[端子状態の表示]** チェック ボックスをオンにします。状態端子がブロックの上に表示されます。

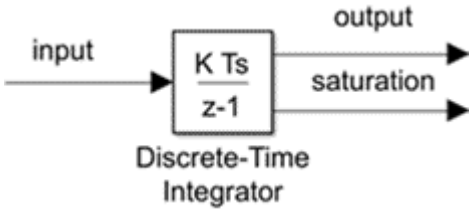


積分の制限

出力が一定のレベルを超えないようにするには、**[出力を制限する]** チェック ボックスをオンにして、対応するテキスト ボックスに制限値を入力します。このようにすると、ブロックは制限付きで積分器として機能します。出力が制限値に達すると、積分動作はオフになり、積分飽和現象を回避します。シミュレーション中、制限値を変更できますが、出力が制限されているかどうかは変更できません。次の表は、ブロックによる出力の決定方法を示しています。

積分	**出力**
[飽和の下限] 以下で入力が負	**飽和の下限**
[飽和の下限] と **[飽和の上限]**	積分
[飽和の上限] 以下で入力が正	**飽和の上限**

状態が制限されていることを示す信号を生成するには、**[飽和端子の表示]** チェック ボックスをオンにします。新しい飽和端子はブロック出力端子の下に表示されます。

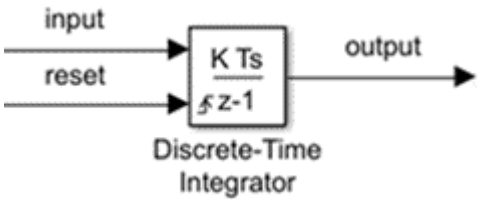


飽和は、次の 3 つの値のいずれかをもちます。

- 1 は、上限が適用されていることを示します。
- 0 は、積分が制限されないことを示します。
- -1 は、下限が適用されていることを示します。

状態のリセット

このブロックは、そのブロックの状態を外部信号に基づいて指定された初期条件にリセットできます。ブロックに状態をリセットさせるには、**[外部リセット]** パラメーターのいずれかのオプションを選択します。リセット トリガーのタイプを示すリセット端子が表示されます。

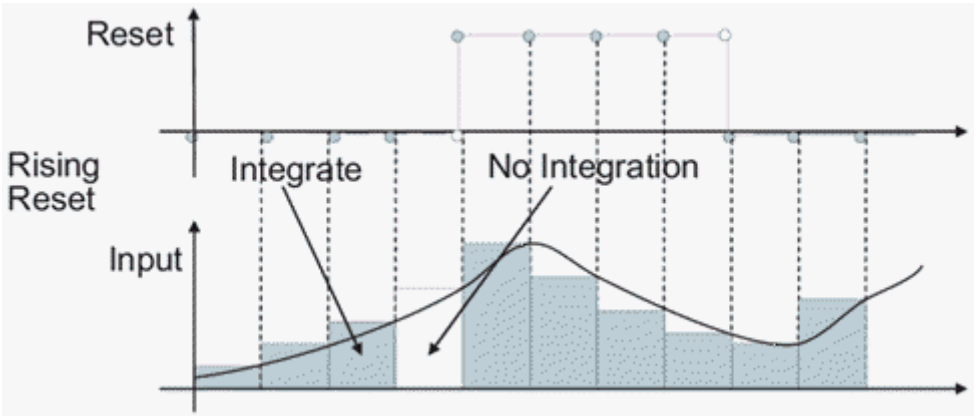


リセット端子は直達をもちます。ブロック出力が直達をもつブロックを通じてこの端子にフィードバックされる場合や、直接フィードバックされる場合、代数ループが発生します。このループを解除するには、ブロックの状態端子の出力をリセット端子に接続します。ブロックの状態にアクセスするには、**[端子状態の表示]** チェック ボックスをオンにします。

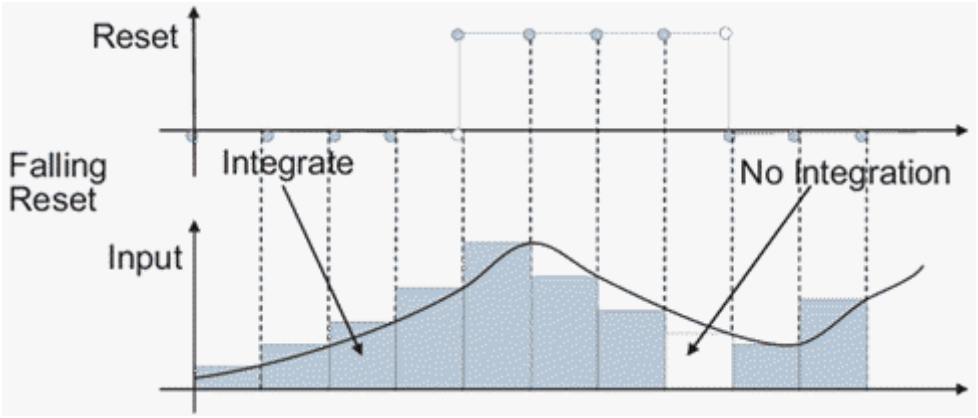
トリガー タイプのリセット

[外部リセット] パラメーターによって、リセットをトリガーするリセット信号の属性を決めることができます。トリガー オプションには以下が含まれます。

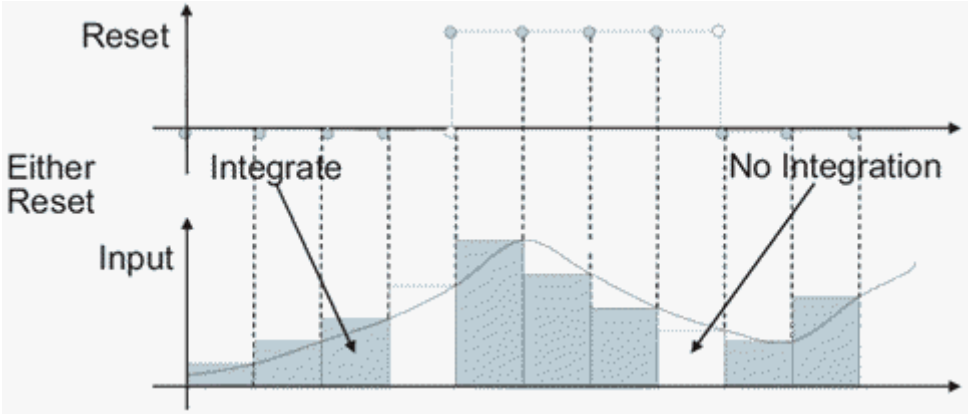
- 立ち上がり – リセット信号が立ち上がりエッジをもつ場合、状態をリセットします。たとえば、次の図は立ち上がりリセット トリガーによる後退オイラー積分の効果を示します。



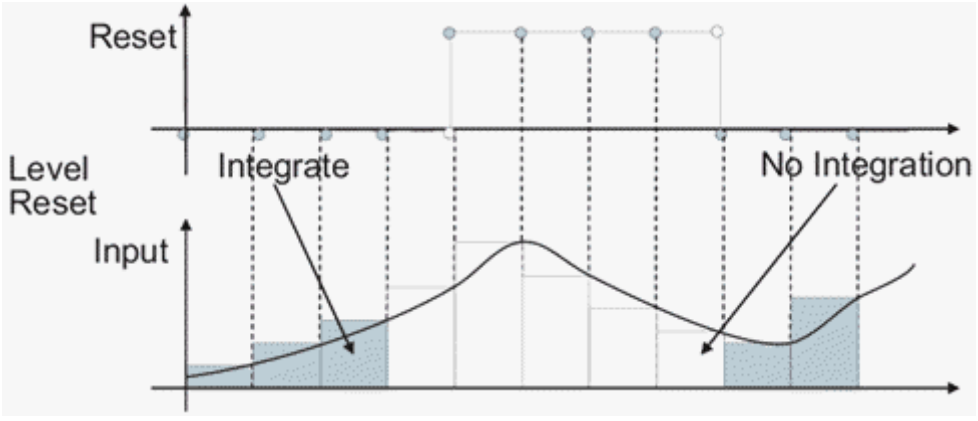
- 立ち下がり – リセット信号が立ち下がりエッジをもつ場合、状態をリセットします。たとえば、次の図は立ち下がりリセット トリガーによる後退オイラー積分の効果を示します。



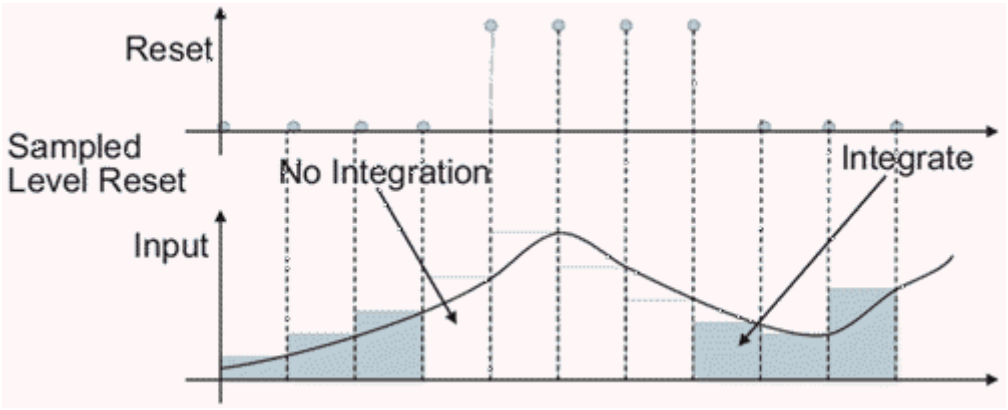
- 両方 – リセット信号が立ち上がりまたは立ち下がりをもつ場合、状態をリセットします。たとえば、次の図はいずれかのリセット トリガーによる後方オイラー積分の効果を示します。



- レベル – リセット信号が非ゼロである間、出力を初期条件にリセットしてホールドします。たとえば、次の図はレベル リセット トリガーによる後退オイラー積分の効果を示します。



- サンプル レベル – リセット信号が非ゼロの場合、出力を初期条件にリセットします。たとえば、次の図はサンプル レベル リセット トリガーによる後退オイラー積分の効果を示します。



[サンプル レベル] リセット オプションは計算量が少ないため、[レベル] リセット オプションよりも効率的です。

> **メモ**
>
> Discrete-Time Integrator ブロックでのトリガーの検出は、いずれも正の値をもつ信号に基づいて行われます。たとえば、-1 から 0 に遷移する信号は立ち上がりエッジと見なされませんが、0 から 1 に遷移する信号は立ち上がりエッジと見なされます。

簡易初期化モードの動作

簡易初期化モードは、[コンフィギュレーション パラメーター] ダイアログで **[指定不足の初期化の検出]** を [簡易] に設定すると有効になります。簡易初期化モードを使用すると、Discrete-Time Integrator ブロックはクラシック初期化モードとは異なった動作をします。新しい初期化動作はロバスト性が高く、以下の場合は整合性のある動作をします。

- 代数ループ内
- イネーブルとディセーブル時
- トリガー サンプル時間を使用した結果と明示的なサンプル時間を比較する場合。なお、ブロックは明示的なサンプル時間と同じレートでトリガーがかかります。

簡易初期化モードを使用すると、Continuous-Time Integrator ブロックを Discrete-Time Integrator ブロックに変換しやすくなります。これは初期条件が両方のブロックで同じ意味をもつためです。

従来の初期化モードと簡易初期化モードの詳細は、指定不足の初期化の検出を参照してください。

[初期条件設定] を [出力] に設定した場合のイネーブルとディセーブルの動作

Triggered Subsystem と Function-call Subsystem の **[初期条件設定]** を [出力] に設定して簡易初期化モードを使用する場合、ブロックのイネーブルとディセーブルの動作は以下のように簡略化されます。

ディセーブル時 t_d:

$$y(t_d) = y(t_d\text{-}1)$$

イネーブル時 t_e:

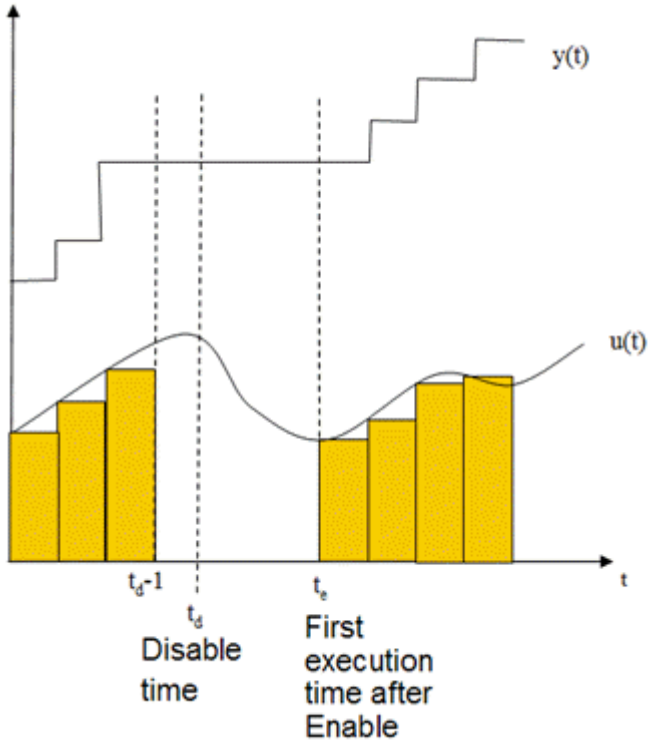
- 親サブシステムの制御端子で **[イネーブル時の状態]** が リセット に設定されている場合:

  $$y(t_e) = IC.$$

- 親サブシステムの制御端子で **[イネーブル時の状態]** が 保持 に設定されている場合:

  $$y(t_e) = y(t_d).$$

  次の図は、この条件を示しています。



反復サブシステム

簡易初期化モードを使用する場合、Discrete-Time Integrator ブロックを反復サブシステム内に置くことはできません。

簡易初期化モードでは、反復サブシステムは経過時間を保持しません。そのため、経過時間を必要とする Discrete-Time Integrator ブロックが反復サブシステムブロック内に置かれた場合、Simulink はエラーを報告します。

Function-Call Subsystem 内の Enabled Subsystem での動作

Discrete-Time Integrator ブロックが含まれる Enabled Subsystem を含む Function-Call Subsystem システムがある場合、次の動作が適用されます。

積分手法	関数呼び出しトリガー端子のサンプル時間タイプ	Function-Call Subsystem をイネーブルにした後に始めて実行するときの `delta T`	動作の理由
前進オイラー法	トリガー	`t – tstart`	Function-Call Subsystem を最初に実行するとき、積分アルゴリズムは `tstart` を以前のシミュレーション時間として使用します。
後退オイラー法および台形則	トリガー	`t – tprevious`	Function-Call Subsystem を最初に実行するとき、積分アルゴリズムは `tprevious` を以前のシミュレーション時間として使用します。
前進オイラー法、後退オイラー法、および台形則	周期的	関数呼び出しジェネレーターのサンプル時間	周期モードでは、Discrete-Time Integrator ブロックは、`delta T` の関数呼び出しジェネレーターのサンプル時間を使用します。

端子

入力

すべて展開する

Port_1 — 入力信号
スカラー | ベクトル | 行列

IC — 状態の初期条件
スカラー | ベクトル | 行列

出力

すべて展開する

Port_1 — 入力の離散時間積分または累積
スカラー | ベクトル | 行列

Port_2 — 飽和の出力
スカラー | ベクトル | 行列

Port_3 — ステートの出力
スカラー | ベクトル | 行列

パラメーター

すべて展開する

メイン

積分手法 — 累積手法
積分：前進オイラー法 (既定値) | 積分：後退オイラー法 | 積分：台形則 | 累積：前進オイラー法 | 累積：後退オイラー法 | 累積：台形則

ゲイン値 — 積分器の入力の乗算に使用する値
`1.0` (既定値) | スカラー | ベクトル

外部リセット — 状態を初期条件にリセットするタイミングの選択
なし (既定値) | 立ち上がり | 立ち下がり | 両方 | レベル | [サンプル レベル]

初期条件のソース — 初期条件のソースを選択
内部 (既定値) | 外部

初期条件 — 状態の初期条件
`0` (既定値) | スカラー | ベクトル | 行列

初期条件設定 — 初期条件を適用する場所を選択
自動 (既定値) | 出力 | 互換性

サンプル時間 (継承は -1) — サンプルの間隔
`-1` (既定値) | スカラー | ベクトル

出力を制限する — ブロックの出力値を指定した範囲に制限
off (既定値) | on

飽和の上限 — 積分の上限
inf (既定値) | スカラー | ベクトル | 行列

飽和の下限 — 積分の下限
-inf (既定値) | スカラー | ベクトル | 行列

飽和端子の表示 — 飽和出力端子を有効化
off (既定値) | on

状態端子の表示 — 状態出力端子を有効化
off (既定値) | on

線形化時に出力制限とリセットの設定を無視 — ブロックのリセットを無効化
off (既定値) | on

Signal Attributes

出力の最小値 — 範囲チェックの最小出力値
[] (既定値) | スカラー

出力の最大値 — 範囲チェックの最大出力値
[] (既定値) | スカラー

データ型 — 出力データ型
継承: 内部ルールによる継承 (既定値) | 継承: 逆伝播による継承 | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | [fixdt(1,16)] | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

固定小数点ツールによる変更に対して出力データ型の設定をロックする — 固定小数点ツールが出力データ型をオーバーライドするのを防止
off (既定値) | on

整数丸めモード — 固定小数点演算の丸めモードを指定
負方向 (既定値) | 正方向 | 最も近い偶数方向 | 最も近い正の整数方向 | 最も近い整数方向 | 最も簡潔 | ゼロ方向

整数オーバーフローで飽和 — オーバーフロー アクションの方法
off (既定値) | on

状態属性

状態名 — ブロックの状態の一意の名前
'' (既定値) | 英数字の文字列

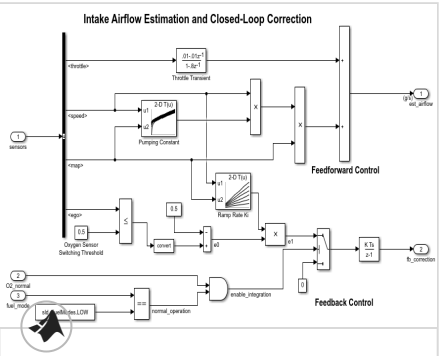
状態名を Simulink の信号オブジェクトに関連付ける — 状態名の信号オブジェクトへの関連付けを要求
Off (既定値) | boolean

信号オブジェクト クラス — カスタム ストレージ クラスのパッケージ名
Simulink.Signal (既定値)

コード生成ストレージ クラス — コード生成のストレージ クラス
自動 (既定値) | モデルの既定の設定 | ExportedGlobal | ImportedExtern | ImportedExternPointer | Bitfield (Custom) | Volatile (Custom) | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

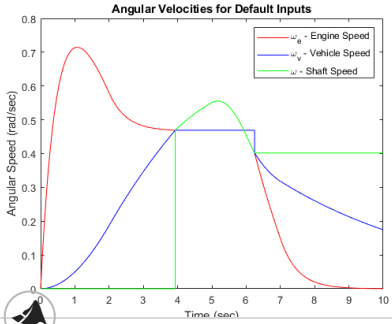
TypeQualifier — ストレージ型修飾子
'' (既定値) | const | volatile | ...

モデルの例



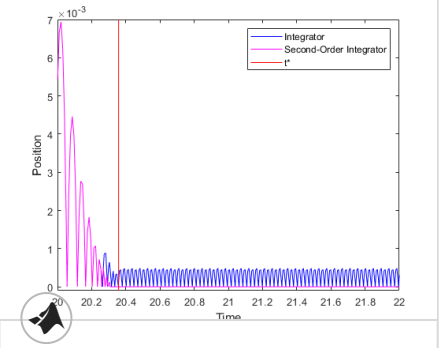
前進オイラー積分法を使用した離散時間積分

Sldemo_fuelsys モデルでは、fuel_rate_control/airflow_calc サブシステムの Discrete-Time Integrator ブロックを使用します。



クラッチ ロックアップ モデルの作成

この例では、Simulink® を使用して回転クラッチ システムをモデル化およびシミュレーションする方法を示します。ロックアップ時にシス



跳ねるボールのシミュレーション

この例では、Simulink® を使用した跳ねるボールのモデル化に対する 2 種類の異なるアプローチの使用方法を示します。

ブロックの特性

データ型	`double` | `fixed point` | `integer` | `single`
直接フィードスルー	はい
多次元信号	いいえ
可変サイズの信号	いいえ
ゼロクロッシング検出	いいえ

拡張機能

› **C/C++ コード生成**
Simulink® Coder™ を使用して C および C++ コードを生成します。

› **HDL コード生成**
HDL Coder™ を使用して FPGA 設計および ASIC 設計のための Verilog および VHDL のコードを生成します。

PLC コード生成
Simulink® PLC Coder™ を使用して構造化テキスト コードを生成します。

固定小数点の変換
Fixed-Point Designer™ を使用して固定小数点システムの設計とシミュレーションを行います。

参考

Integrator

トピック

個別の信号、状態、およびパラメーター データ要素へのストレージ クラスの適用 (Simulink Coder)
個別の信号、状態、およびパラメーター データ要素へのストレージ クラスの適用 (Simulink Coder)
Apply Built-In and Customized Storage Classes to Data Elements (Embedded Coder)

R2006a より前に導入