Kyushu Institute of Technology

平成 18 年度 卒業論文

XMLバインディングを用いた 制御系のモデリング・ シミュレーション プラットフォームの開発

Development of Modeling and Simulation Platform for Control System with XML Binding

2007年2月27日

指導教員
古賀 雅伸 助教授

提出者
九州工業大学 情報工学部 制御システム工学科
学籍番号　03233068
氏　　名　松本 明紘

目 次

図 目 次

表 目 次

第 1 章

序論

本章では研究の背景と目的について述べる。

1.1　背景

ソフトウェア開発において，要求仕様がめまぐるしく変化するにもかかわらず迅速な開発を可能にするアジャイル開発手法が注目を集めている [4]。アジャイル (Agile) とは，俊敏という意味である。アジャイル開発のために優れた開発支援ツールも開発されている。制御系の開発においても同様のことが言え，制御対象が複雑化しているにもかかわらず，制御器に対する要求は厳しくなる上に納期短縮が求められており迅速な開発が求められている。

制御系の開発プロセスでは，**図 1.1** に示すような制御対象のモデリング，解析，制御器の設計，シミュレーションというプロセスを繰り返し行う。制御対象が年々巨大化していく中で使いやすいモデリング · シミュレーション支援ツールへの需要は大きなものとなっている。

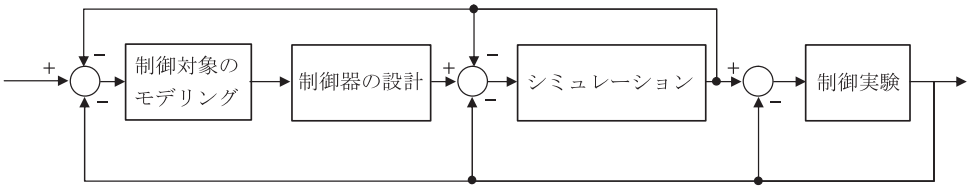


図 1.1 制御系開発プロセス

一般に制御システムのモデリングでは，伝達関数表現や状態空間表現などの数式モデルが使用される。複雑なシステムのモデリングでは，それを構成する各部分システムの数式モデルと部分システムの結合関係でモデルが表現される。この部分システムの結合関係を視覚的に表現するために**ブロック線図**が使用される。

一般的なモデリングからシミュレーションの流れを**図 1.2** に示した。まず，制御対象をブロック線図でモデリングし，シミュレーションプログラムの作成，シミュレーションという作業を行う。

シミュレーションプログラムの作成は MATX[5] や Matlab[6] などの数値計算言語を用いて行う。このとき，制御系の開発者には数値計算言語の専門知識も必要になるため大きな負担となってしまう。シミュレーションプログラムの作成工程なしで，ブロック線図による設

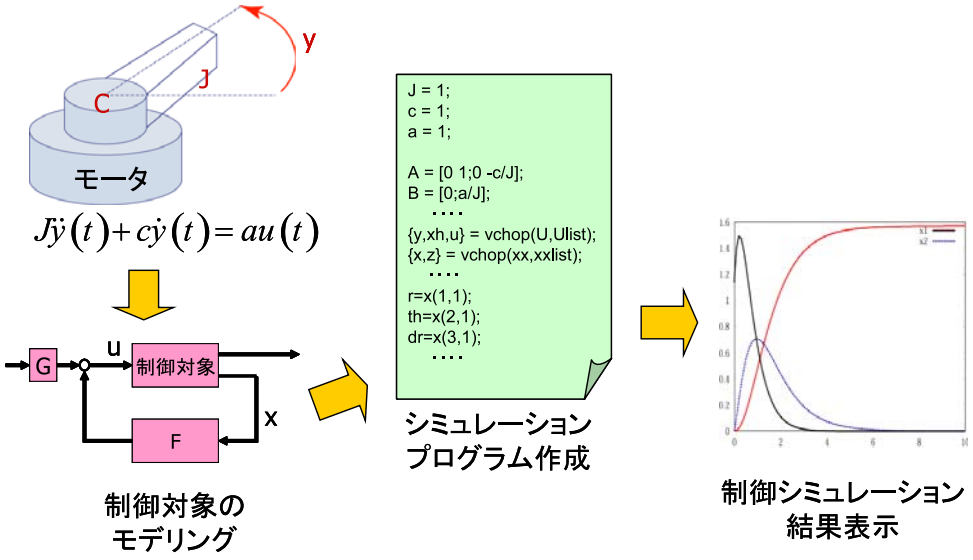


図 1.2 モデリングからシミュレーション

計だけでシミュレーションを行うことができれば開発者の負担を大きく軽減することができる。

1.2 目的

本研究では，制御系の開発プロセスを効率化できるフリーの制御系のモデリング · シミュレーションツール Jamox(Java Agile MOdeling Tool for Control System) の開発を目的としている。

以下にツールの主な機能を挙げる。

- 簡単なマウス操作でブロック線図を作成
- ブロック線図のデータは XML として保存
- システムの伝達関数，状態空間表現導出
- ボード線図等の周波数応答シミュレーション
- ステップ応答等の時間応答シミュレーション

Jamox は，環境に依存しない Java 言語で開発され，フリーソフトウェアとして配布される予定である。

Jamox はプラグインアーキテクチャを採用しており，プラグインを導入することで開発をさらに有効に進めることができる。プラグインを自分で開発することもでき，必要な処理をプラグインすることで様々な用途に利用可能になる。また，数値計算ライブラリにはデフォルトでは NFC(Numerical Foundation Classes)[7] を，グラフ表示には JPit[8] を用いており，このライブラリを変更することも可能である。

1.3 関連研究

ブロック線図ベースのモデリングツールとして商用製品である Simulink[9]，Dymola[10]，WinCSCAD[11]，フリーソフトでは，Scicos[12]，BrainBox[13] などがある。

Simulink

Simulink は関数ブロックや伝達関数ブロックによってシステムをモデル化することができ，メカニカル，電気系等，汎用的に適用が可能である。また，連続系 · 離散系の様々な数式表現を取り扱い非線形システムにも対応しているなどの特徴を有するが，非常に高額である。研究者が開発したツールを無料で配布しても，製品本体を購入しなければそのツールを使用できないという問題がある。また，ソースコードが公開されていないため，何か問題を見つけたとしても自分たちで修正することはできず，開発者が修正してくれるのを待つしかない。Simulink の外観を**図 1.3** に示す。

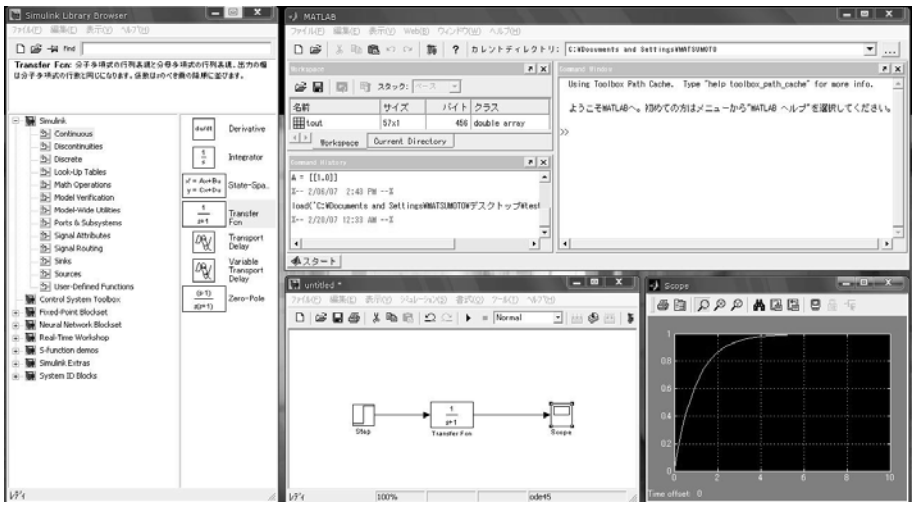


図 1.3 Simulink の外観

Modelica

Dymola はスウェーデンの Dynasim AB 社によって開発された部品ベースモデラーである。豊富な部品ライブラリで，メカニクス，電気回路，モータなどが複合する解析対象のモデリングとシミュレーションが可能である。また，上で紹介した Simulink との連携が可能である。しかし，適用範囲が広いため制御系開発に特化しているわけではなく，制御系の開発には不向きである。Dymola の外観を**図 1.4** に示す。

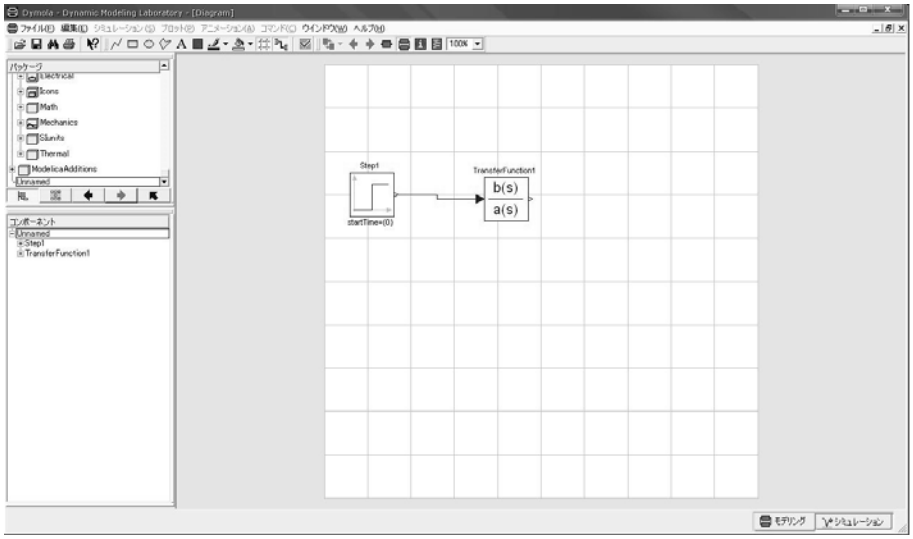


図 1.4 Dymola の外観

Scicos

Scicos は Scilab のツールボックスであり，フランス国立コンピュータ科学 · 制御研究所で開発され，フリーソフトウェアとして公開されている。Scilab では，連続時間系，離散時間系を含む力学系のモデリングおよびシミュレーションを行うことができる。Scicos はブロック線図の配置などの操作がやりづらいという評価がある。Scicos の外観を**図 1.5** に示す。

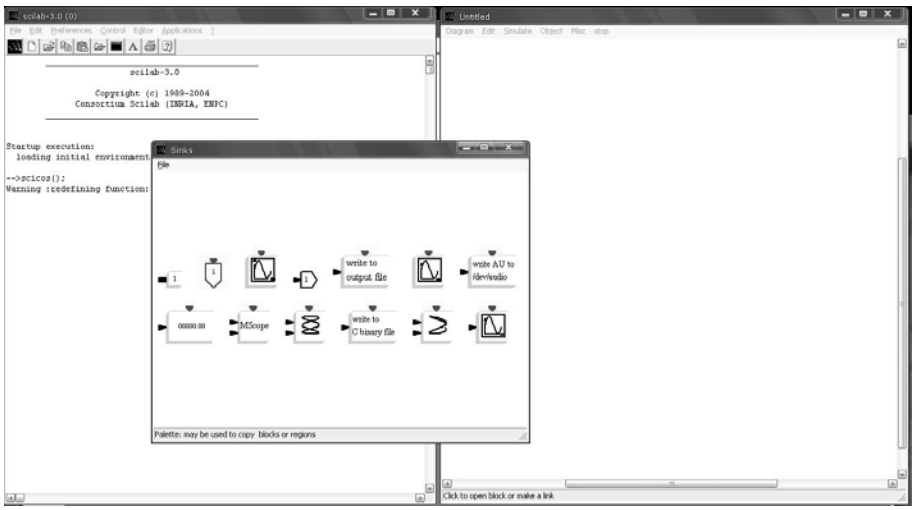


図 1.5 Scicos の外観

WinCSCAD

WinCSCAD は教育用途に開発された制御系 CAD システムで，ステップ応答などの時間応答シミュレーション，ボード線図などの周波数応答シミュレーションなどを行うことができる。その他にもナイキスト線図やニコルズ線図などの表示も行うことができ教育目的には有用である。しかし，取り扱えるシステムが伝達関数表現可能なシステムに限られているため制御系の開発には不向きである。WinCSCAD の外観を**図 1.6** に示す。

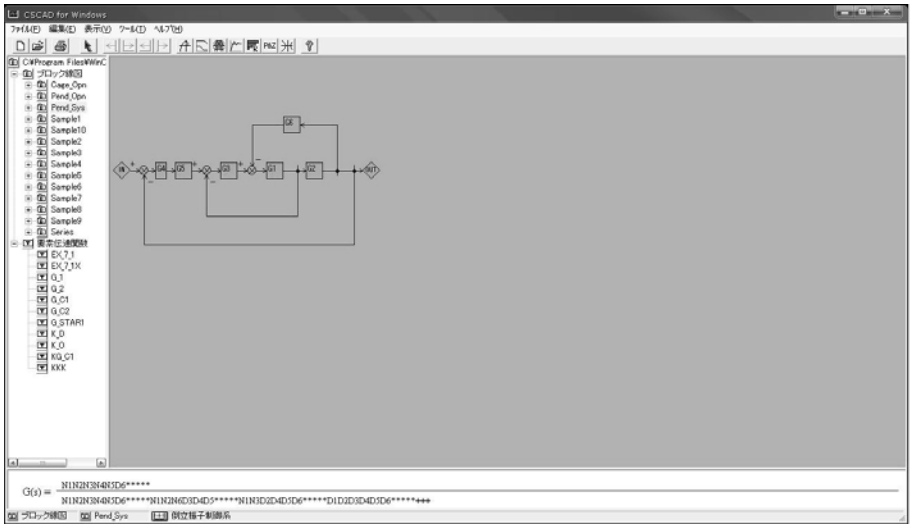


図 1.6 WinCSCAD の外観

BrainBox

BrainBox は，一般的な科学的目的のためのシミュレーションツールボックスである。Eclipse RCP[14] に基づく，フリーのオープンソース・アプリケーションであるため，柔軟性と拡張性が高い。しかし，取り扱えるシステムが少なく，操作も若干やりづらい。BrainBox の外観を**図 1.7** に示す。

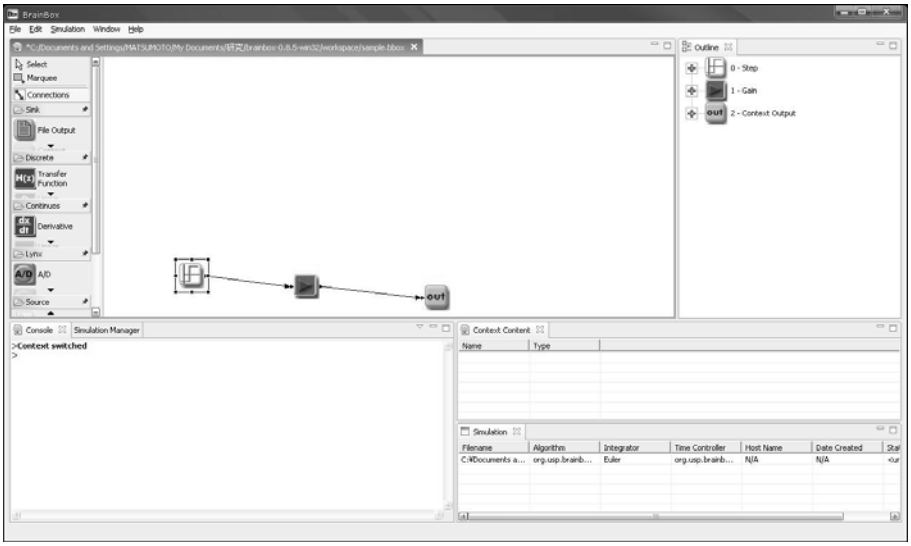


図 1.7 BrainBox の外観

第2章

制御系シミュレーション

2.1　システムの種類

制御システムはいくつかの特徴で大分することができる。以下に各分類について述べる。制御系を構成する各要素は**図 2.1** に示すように，入力 $u(t)$ が加えられるとそれに対応した

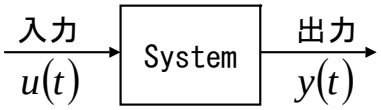


図 2.1 基本システム

出力 $y(t)$ が存在する。この場合の入出力に対して，時間における出力が，その時刻における入力だけによって決定されるものを**静的システム**という。また，現在の状態が過去の状態及び入力に依存して変動するシステムを**動的システム**あるいはダイナミックシステムという。

静的システムは動的システムの特殊な場合のシステムと考えることができるので，動的システムを対象にすれば良い。

我々が表現したいシステムの多くは微分方程式を用いて記述される。その場合，現在時刻の出力 $y(t_0)$ は現在の入力 $u(t_0)$ だけでなく過去の履歴にも依存するのである。

線形システムと非線形システム

ダイナミカルシステム

$$\begin{aligned} & a_n\frac{d^n y(t)}{dt^n} + a_{n-1}\frac{d^{n-1}y(t)}{dt^{n-1}} + \cdots + a_1\frac{dy(t)}{dt} + a_0 y(t) \\ = \quad & b_m\frac{d^m u(t)}{dt^m} + b_{m-1}\frac{d^{m-1}u(t)}{dt^{m-1}} + \cdots + b_1\frac{du(t)}{dt} + b_0 u(t) \end{aligned} \tag{2.1}$$

において，次のような線形性があることは重要である。いま，任意の時刻 t において，システムにある入力 $u_1(t)$ を加えたときに生ずる出力を $y_1(t)$ とし,，また別の入力 $u_2(t)$ を加えたときに生ずる出力を $y_2(t)$ とする。このとき任意の実数 α, β に対して入力 $\alpha u_1(t) + \beta u_2(t)$ を考え，これを加えたときに結果として生ずる出力が $\alpha y_1(t) + \beta y_2(t)$ となる。この性質は重ね合わせの原理と呼ばれており，制御系の解析や設計において重要な役割を果たすことが

知られている。この性質が成り立つシステムは**線形システム**，成り立たない場合は**非線形システム**である。

連続時間システムと離散時間システム

- 連続時間システム

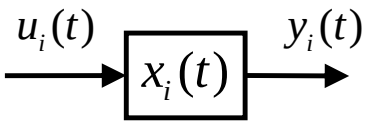


図 2.2 連続時間システム

連続時間システムは，時間的にも数値的にも連続である信号を取り扱うシステムである。今，システムが $x_1(t), x_2(t), \ldots, x_n(t)$ の状態を持っているとするとき，個々のシステムは線形，非線形にかかわらず，連続な時間変数 t の関数として以下のように表現できる。

$$\dot{x}_i(t) = f_i(t, x_i(t), u_i(t)) \tag{2.2}$$
$$y_i(t) = g_i(t, x_i(t), u_i(t) \tag{2.3}$$

ただし，$(i = 1\text{〜}n)$ である。

ここで，n 個の状態を $x(t) = [x_1(t), x_2(t), \ldots, x_n(t)]^T$ とおき，システム全体の入力を $u(t)$，出力を $y(t)$ とおくと，全体の状態方程式，出力方程式は以下のように表すことができる。

$$\dot{x}(t) = F(t, x(t), u(t)) \tag{2.4}$$
$$y(t) = G(t, x(t), u(t)) \tag{2.5}$$

- 離散時間システム

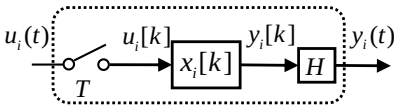


図 2.3 離散時間システム

離散時間信号は時間的に不連続な信号であるが，この信号は**図 2.3** のように，連続時間信号に対し，サンプラーでサンプリング周期 T ごとに値を取得することで得られる。例えば連続時間信号 $y(t)$ に対して，時刻 $t = kT$ における変換値を $y[k]$ と表す。

このとき，図中の破線で囲んだ部分が離散時間システムである。以後，離散時間システムの状態を $\bar{x}[k]$ と表すことにする。

今，システムが $\overline{x_1}[k], \overline{x_2}[k], \ldots, \overline{x_n}[k]$ の状態を持っているとするとき，個々のシステムの状態方程式，出力方程式は以下のように表される。

$$\bar{x}_i[k+1] = f_i(k, \overline{x_i}[k], u_i[k]) \tag{2.6}$$
$$y_i[k] = g_i(k, \overline{x_i}[k], u_i[k]) \tag{2.7}$$

ただし，$(i = 1\sim n)$ である。

ここで，n 個の状態を $\bar{x}[k] = [\bar{x}_1[k], \bar{x}_2[k], \ldots, \bar{x}_n[k]]^T$ とおき，システム全体の入力を $u[k]$，出力を $y[k]$ とおくと，全体の状態方程式，出力方程式は以下のように表すことができる。

$$\bar{x}[k+1] = F(k, \bar{x}[k], u[k]) \tag{2.8}$$
$$y[k] = G(k, \bar{x}[k], u[k]) \tag{2.9}$$

サンプル値制御系

サンプル値制御系とは，連続時間で動作する制御対象をサンプラーとホールドを通して離散時間で動作する離散時間制御器で制御を行う制御系である [15]。ここでは，サンプル点間の挙動も含め，時間的離散性についてはありのままに扱う。**図 2.4** は，単一フィードバックの場合の概念図である。図中の実線の矢印が連続時間信号を表し，点線の矢印が離散時間信号を表す。

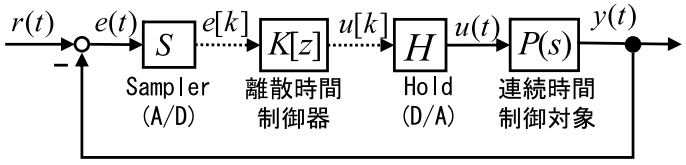


図 2.4 単一フィードバックサンプル値制御系

ここで，$\mathcal{S}$ はサンプラーであり，一定時間間隔 T ごとに連続時間信号 $e(t)$ の値を取り出し，離散時間コントローラ $K[z]$ に入力として $e[k]$ を渡す働きをする。計算機制御では AD 変換器がその役割を担う。一方 $\mathcal{H}$ はホールド要素であり，同じく T の時間間隔ごとに制御器の出力 $u[k]$ をホールドし，連続時間信号として制御対象 $P(s)$ への制御入力とする。こち

らは DA 変換器が行う。最も簡単かつよく用いられるのはサンプル時刻 kT, $k = 0, 1, 2, \ldots$ の値を取り出す理想サンプラー

$$e[k] = \mathcal{S}\{e(t), k\} \equiv e(kT), \qquad k = 0, 1, 2, \ldots \tag{2.10}$$

と，次のサンプル時点まで値を定値に保存する 0 次ホールド

$$u(t) = \mathcal{H}\{u[k], t\} \equiv u(k), \qquad kT \leq t < (k+1)T \tag{2.11}$$

である。サンプリングとホールドのタイミングは同期しているものとするのが普通である。無論，現実にはこれらが，特に理想サンプラーがそのまま実現できるわけではないが，理論的簡明さからよく採用される。

図 2.5 単一フィードバックサンプル値制御系

サンプル値系では連続時間システムと離散時間システムが混在するが，入力信号 $u(t)$ ，出力信号 $y(t)$ は共に連続時間信号であることに注意する。また，連続な時間変数 t と離散時間 (kT) は

$$t = kT$$

の関係があるので，その時の時刻 t とサンプリング周期 T が分かれば k が求まることがわかる。また，システムの状態を知るには，連続時間システムと離散時間システムの両方について調べる必要がある。

ここで，システム全体の状態方程式，出力方程式を以下のように表すことができる。

$$\dot{x}(t) = F(t, x(t), \bar{x}[k], u(t)) \tag{2.12}$$

$$\bar{x}[k+1] = H(t, x(t), \bar{x}[k], u(t)) \tag{2.13}$$

$$y(t) = G(t, x(t), \bar{x}[k], u(t)) \tag{2.14}$$

このとき，式中の $x(t), \bar{x}[k]$ は，それぞれサンプル値系中に存在する全ての連続時間システム，離散時間システムの状態により構成されるベクトルである。また，離散時間システムの状態を表すベクトル $\bar{x}[k]$ 中に n 個の要素があるとき，それぞれのサンプリング周期 T_i が必ずしも全て等しいとは限らないので注意が必要である。

2.1.1 伝達関数

微分方程式で記述される線形ダイナミカルシステムは，その表現のままでは入出力関係に対する見通しが必ずしも良いとは言えない。制御系の解析や設計においては，複数のシステムの結合を扱う必要がしばしば生じるが，以下のような困難がある。

次式のように二つのシステムが直列に結合された場合を考える。$u_1(t)$ は (2.15) 式で記述されるシステムに加えられる入力であり，その出力 $y_1(t)$ が (2.16) 式のシステムに直接入力される。これに対応する出力が $y_2(t)$ である。

$$\frac{d^2y_1(t)}{dt^2} + 2\frac{dy_1(t)}{dt} + 3y_1(t) = \frac{du_1(t)}{dt} + u_1(t) \tag{2.15}$$

$$\frac{d^2y_2(t)}{dt^2} + 3\frac{dy_2(t)}{dt} = 2\frac{dy_1(t)}{dt} + y_1(t) \tag{2.16}$$

このとき，たとえ個々のシステムにおける入出力関係がわかったとしても，上記の連立微分方程式表現のままでは，結合システム全体における入力 $u_1(t)$ と出力 $y_2(t)$ の関係を把握することは難しい。全体を一つの微分方程式で書き直すことは可能であるが，これも微分方程式表現のままでは簡単とは言えない。

ここで，線形ダイナミカルシステムにおいて全ての初期値を 0 としたときの，出力のラプラス変換と入力のラプラス変換の比，すなわち

$$\text{伝達関数} = \frac{\text{出力のラプラス変換}}{\text{入力のラプラス変換}}$$

によって与えられる**伝達関数**という概念を用いて表現すると，上述のような問題点が解消され，かつ表現も簡潔なものとなる。

(2.1) 式の伝達関数を求めるには，まず，(2.1) 式において全ての初期値を 0 とし，出力 $y(t)$ と入力 $u(t)$ のラプラス変換を，それぞれ $\mathcal{L}[y(t)] = Y(s)$,$\mathcal{L}[u(t)] = U(s)$ とおく。(2.1) 式の両辺をラプラス変換すれば

$$\begin{aligned} & (a_ns^n + a_{n-1}s^{n-1} + \cdots + a_1s + a_0)Y(s) \\ = \; & (b_ms^m + b_{m-1}s^{m-1} + \cdots + b_1s + b_0)U(s) \end{aligned} \tag{2.17}$$

を得る。よって，いまの伝達関数を $G(s)$ とすると

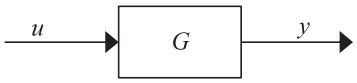
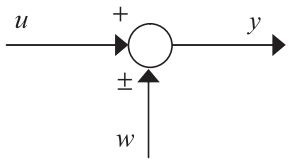
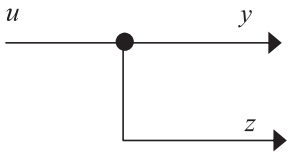
$$\begin{aligned} G(s) &= \frac{Y(s)}{U(s)} \\ &= \frac{b_ms^m + b_{m-1}s^{m-1} + \cdots + b_1s + b_0}{a_ns^n + a_{n-1}s^{n-1} + \cdots + a_1s + a_0} \end{aligned} \tag{2.18}$$

となる。これは微分方程式と比較すると，簡潔な表現となっている。また伝達関数の定義から $Y(s) = G(s)U(s)$ となるので，出力のラプラス変換 $Y(s)$ は伝達関数 $G(s)$ と入力のラプラス変換 $U(s)$ との積で求まることがわかる。

2.2 ブロック線図

システムを構成する一つの要素をブロックで表現し，サブシステムの関係をブロックのつながりという概念でとらえたものがブロック線図である。信号に対する変換や演算を表すために，つぎの三つの基本単位がある。

表 2.1 ブロック線図の基本単位

基本単位	記号	式
ブロック		$y = Gu$
加え合わせ点		$y = u \pm w$
引き出し点		$y = u, z = u$

1. **ブロック**
   表 2.1 の上段にある記号に示したように，伝達特性 G をブロックで囲んで，入出力関係の式 $y = Gu$ を示す。ブロックの中には，主として伝達か数を記述する。ただし，その他の入出力関係を示す式またはグラフなどが書かれる事もある。

2. **加え合わせ点**
   表 2.1 の中段にある記号は，二つの信号が加算，あるいは減算されることを示している。したがって，和 $u = u + w$(差の場合は $y = u - w$) という代数的な関係を図を用いて表していることになる。

3. **引き出し点**
   表 2.1 の下段にある記号は，信号 u が分岐する様子を図式的に示している。この信号は分流ではなく抽出なので元の信号 u に対して分岐をいくつ行っても，u には影響を及ぼさない。

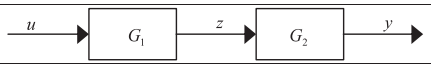
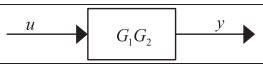
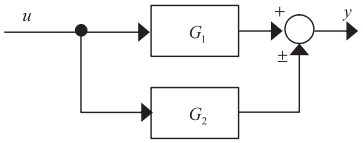
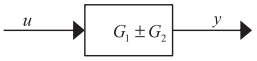
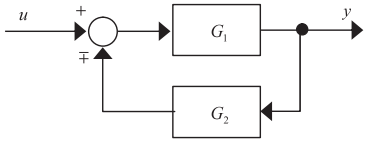
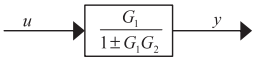
これらの基本単位を複数組み合わせることでブロック線図は形成される。ブロック線図は結合演算を行うことができる。基本的な統合方式を**表 2.2** に示す。

2.2.1 ブロック結合

1. **直列結合**
   表 2.2 の直列結合の行では，伝達関数で表された二つのブロックが直列に接続された場合を表している。伝達関数の定義から $y = G_2 z$, $z = G_1 u$ であることから z を消去して，$y = G_1 G_2 u$ を得る。

2. **並列結合**
   表 2.2 の中段は，並列に接続された二つの伝達関数のブロックを表している。同様に

表 2.2 ブロック線図の結合方式

結合方式	結合前	結合後
直列結合		
並列結合		
フィードバック結合		

して $y_1 = G_1 u$, $y_2 = G_2 u$ であり，また $y = y_1 \pm y_2$ であるから，$y = (G_1 \pm G_2)\, u$ を得る。

3. **フィードバック結合**
表 2.2 の下段のようにブロックが結合されているとき，これをフィードバック結合という。特に加え合わせ点において，一方の符号が負であれば**ネガティブ・フィードバック**といい，正であれば，**ポジティブ・フィードバック**という。

フィードバック経路の伝達関数 G_2 の出力を z とおくと，$z = G_2 y$,$y = G_1 (u - z)$ となっている。これより z を消去すると

$$y = G_1 (u - G_2 y)$$
$$(1 + G_1 G_2) = G_1 u$$

であるから，次式を得る。

$$y = \frac{G_1}{1 + G_1 G_2} u$$

3 つ以上のブロックは上記の結合を繰り返し適用することで簡単化することが可能である。

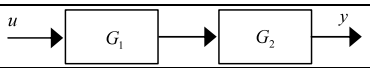
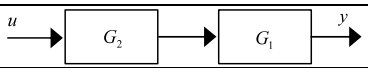
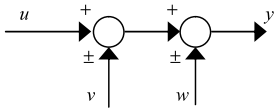
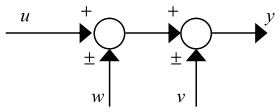
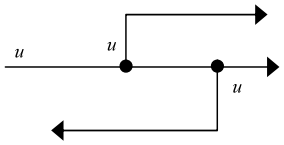
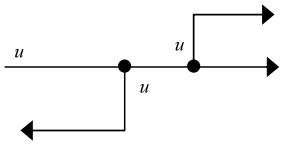
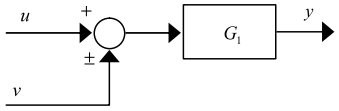
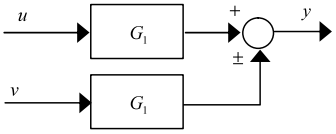
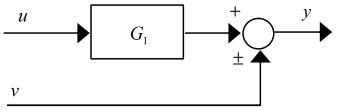
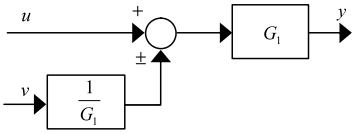
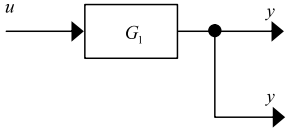
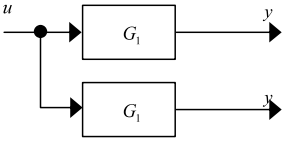
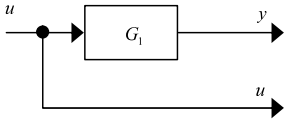
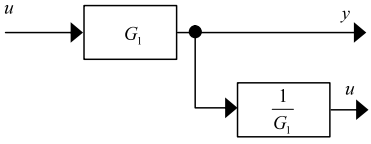
2.2.2 等価変換

表 2.3 にブロック線図で行うことが可能な等価変換を示した。加え合わせ点と引き出し点を入れ替えることができないことに注意する。

2.3 制御シミュレーション

制御系のシミュレーションは，常微分方程式群の数値積分により行う。数値積分のアルゴリズムはいくつかあり，全てのシステムに対して効率よく，正確にシミュレーションを行う

表 2.3 ブロック線図の等価変換

等価変換	変換前	変換後
ブロックの入れ替え		
加え合わせ点の入れ替え		
引き出し点の入れ替え		
ブロックと加え合わせ点の入れ替え(1)		
ブロックと加え合わせ点の入れ替え(2)		
ブロックと引き出し点の入れ替え(1)		
ブロックと引き出し点の入れ替え(2)		

ことができるアルゴリズムというものはないので積分手法の選択やシミュレーションパラメータの設定を行わなければならない。

2.3.1 積分手法

積分アルゴリズムのうち代表的な Runge-Kutta 法について説明を行う。

4 次の Runge-Kutta 法

Runge-Kutta 法は時間が Δt 進む間、x,y の傾き (変化量) の修正を行いながら，その平均を求めて，誤差を少なくする方法である．1 階の常微分方程式

$$\frac{dx}{dt} = f(t, x)$$

において，t が t_0 から h だけ増加した点 $t_1 = t_0 + h$ における値 x_1 を以下の計算で求める。

$$\begin{aligned} k_1 &= h * f(t_0, x_0) \\ k_2 &= h * f(t_0 + \frac{h}{2}, x_0 + \frac{k_1}{2}) \\ k_3 &= h * f(t_0 + \frac{h}{2}, x_0 + \frac{k_2}{2}) \\ k_4 &= h * f(t_0 + h, x_0 + k_3) \end{aligned}$$

として，

$$x_1 = x_0 + \frac{(k_1 + 2 * k_2 + 2 * k_3 + k4)}{6}$$

上記の計算により微分方程式の解 x_1 が求まる。

Runge-Kutta-Fehlberg 法

前述の Runge-Kutta は 4 段であるが，Runge-Kutta-Fehlberg は 6 段であり 5 次の収束である。これらの関数値のうちの 4 個と適当な係数を組み合わせて 4 次の公式をつくり，また 6 個全部と別の適当な係数を組み合わせて 5 次の公式をつくる。これから得られる 2 個の値を比較して誤差評価を行い，ステップ幅の調整に使う。この方法は非常に計算回数を節約することができ，しかも誤差は刻み幅の 5 乗に比例するので 4 次の Runge-Kutta より更に小さくすることができる。

2.3.2 直達項のループ

直達項 (Direct-Path) のループは，自分自身の出力をフィードバックにより，直接フィードフォワードするような場合に生じる。直達項のループが存在する場合，各時刻 t での解が存在するか繰り返し計算を行う必要がある。直達項を持つシステムには以下のようなシステムなどがある

- ゲインシステム
- ほとんどの非線形システム
- 分子と分母の次数が等しい伝達関数
- D 行列がゼロでない状態空間表現システム

図 2.6 に直達項のループが存在するシステムの例を示す。各時刻 t での解を求めるための手法が Newton 法・Newton-Raphson 法である。

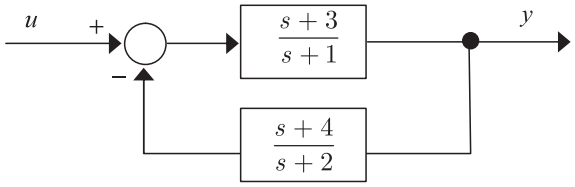


図 2.6 直達項のループが存在するシステム

Newton 法

方程式 $f(x) = 0$ と初期近似値 x_0 が与えられたとする。点 $(x_0, f(x_0))$ を通り，傾き $\dot{f}(x_0)$ である直線の方程式を求めると

$$y - f(x_0) = f\dot{(}x)(x - x_0)$$

ここで x 軸との交点の x 座標を近似解 x_1 とすると，

$$x_1 = x_0 - \frac{f(x_0)}{\dot{f}(x_0)}$$
$$x_n = x_{n-1} - \frac{f(x_{n-1})}{\dot{f}(x_{n-1})}$$

によって逐次的に解を改良していく方法を Newton 法という。

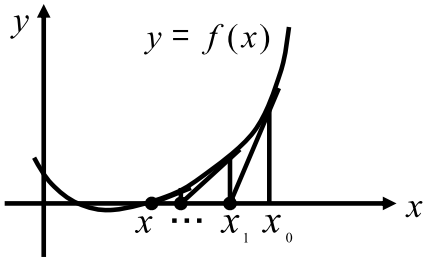


図 2.7 ニュートン法

Newton-Raphson 法

Newton 法は次のような連立非線形方程式の実数解を求める場合に拡張することができる。

$$u = f_1(x, y) \qquad v = f_2(x, y)$$

について，どちらも 0 になる点を求めることを考える。これは $x - y$ 平面上での 2 つの曲線の交点を求めることに対応する。

解に近い点 (x_0,y_0) が与えられたとする。点 (x_0,y_0) における Taylor 展開を求める。(x,y) は (x_0,y_0) に十分近いものとして $(x-x_0)^2$ や $(y-y_0)^2$ などの項を省略すると

$$\begin{aligned} u &\simeq f_1(x_0+\delta x, y_0+\delta y) \\ &= f_1(x_0,y_0) \\ &+ \frac{\partial}{\partial x} f_1(x_0,y_0)(x-x_0) + \frac{\partial}{\partial y} f_1(x_0,y_0)(y-y_0) \end{aligned}$$

$$\begin{aligned} v &\simeq f_2(x_0+\delta x, y_0+\delta y) \\ &= f_2(x_0,y_0) \\ &+ \frac{\partial}{\partial x} f_2(x_0,y_0)(x-x_0) + \frac{\partial}{\partial y} f_2(x_0,y_0)(y-y_0) \end{aligned}$$

となる。これは

$$\begin{aligned} \begin{bmatrix} u \\ v \end{bmatrix} &= \begin{bmatrix} f_1(x_0,y_0)) \\ f_2(x_0,y_0) \end{bmatrix} \\ &+ \begin{bmatrix} \frac{\partial}{\partial x} f_1(x_0,y_0) & \frac{\partial}{\partial y} f_1(x_0,y_0) \\ \frac{\partial}{\partial x} f_2(x_0,y_0) & \frac{\partial}{\partial y} f_2(x_0,y_0) \end{bmatrix} \begin{bmatrix} x-x_0 \\ y-y_0 \end{bmatrix} \end{aligned}$$

と表せる。ここでヤコビ行列 J を

$$J(x_0,y_0) := \begin{bmatrix} \frac{\partial}{\partial x} f_1(x_0,y_0) & \frac{\partial}{\partial y} f_1(x_0,y_0) \\ \frac{\partial}{\partial x} f_2(x_0,y_0) & \frac{\partial}{\partial y} f_2(x_0,y_0) \end{bmatrix}$$

とおくと，右辺が 0 となるとき

$$\begin{aligned} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} &:= \begin{bmatrix} x-x_0 \\ y-y_0 \end{bmatrix} \\ &= -J^{-1}(x_0,y_0) \begin{bmatrix} f_1(x_0,y_0) \\ f_2(x_0,y_0) \end{bmatrix} \end{aligned}$$

となる。このような点 $(x,y) = (x_0,y_0) + (\Delta x, \Delta y)$ を新しい近似解とすることで 2 変数の Newton 法が得られる。これを Newton-Raphson 法という。

第3章

Jamoxの概要

本章では開発したツールである Jamox について述べる。

3.1　Jamoxとは

Jamox は Java Agile MOdeling Tool for Control System の略で Java 言語で開発されたブロック線図ベースのフリーのモデリング·シミュレーションツールである。GUI には SWT[付録 A] を用いており，各 OS(Windows，Linux，Machintosh) のネイティブな GUI を使用するため軽快かつ操作性が良い。

ユーザはブロック線図を描くことでシステムをモデリングし，ソルバとシミュレーションパラメータを選択するだけで簡単にシミュレーションを行うことができる。そのため，制御系の設計者に対する負担を減らすことができる。

3.2　Jamoxの特徴

制御系の開発者はモデリング·シミュレーションツールに対して以下のような需要を持っている。

- ブロック線図を簡単に作成したい
- 以前作成したシステムを再利用したい
- システムが妥当であるか検証したい
- システムの任意の点から任意の点への伝達関数を求めたい
- ボード線図等の周波数応答解析を行いたい
- ステップ応答等の時間応答解析を行いたい
- シミュレーション応答を自作ツールで利用したい
- シミュレーション環境を低コストで構築したい

Jamox はフリーソフトウェアなので，計算機さえあれば無料でシミュレーション環境を構築することが可能である。また，Java 言語で開発されていることからマルチプラットフォームで動作する[1]。Jamox の主な機能を次に示す。

[1]SWT を使用しているため Windows, Linux, Solaris8, QNX, AIX, HP-UX, Mac OSX 上で動作するがサポートするバージョンに限りがある。http://www.eclipse.org/swt/を参照

- 簡単なマウス操作でブロック線図を作成
- ブロック線図のデータを XML として保存，呼出
- ブロック線図の図を画像などに出力
- モデルの整合性検証
- システムの伝達関数結合演算
- ボード線図等の周波数応答シミュレーション
- ステップ応答等の時間応答シミュレーション

Jamox は以下のようにして起動する。また必要なライブラリには classpath が通っている必要がある。

```
% java org.mklab.jamox.Jamox
```

Jamox の実行画面を**図 3.1** に示す。ツールによって提供されるブロックをライブラリからキャンバスにドラッグ&ドロップすることでブロックを配置することができる。

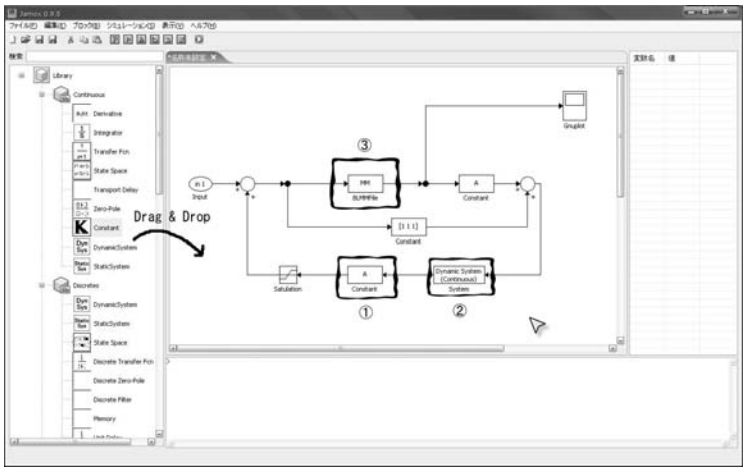


図 3.1 Jamox の利用画面

それぞれのブロックはマウス操作で接続することができ，簡単にブロック線図を画面上に作成することができる。**図 3.1** に示す①はライブラリに登録されている基本ブロックであり，システムの種類や構造は決まっており，パラメーターだけが変更可能である。②は任意の動的システムや静的システムを表すユーザ定義ブロックであり，*DynamicSystem* クラスや *StaticSystem* クラスを継承した Java クラスを対応付ける。制御システムの分類を参考にクラスの継承を利用することで効率的に再利用性の高いブロックを作成できる。また，数値計算ライブラリ NFC[7] を用いることで簡単に高速な計算コードが記述できる。③は MATX[5] 文法に従って記述されたシステムを取り扱うブロックであり，既存の MATX コードを再利用できる。

最も頻繁に使用される線形連続時間システムだけでなく，一般的なシステムを表現する非線形連続時間システムも取り扱える。また，離散時間形表現の線形システムと非線形システムにも対応しており，制御で利用されるほとんどのシステムを取り扱える。

3.3 シミュレーションプラットフォーム

プラットフォーム (platform) とは基盤，土台などという意味である。本ツール Jamox の位置づけはシミュレーションのための基盤である。Jamox はブロック線図を作成後シミュ

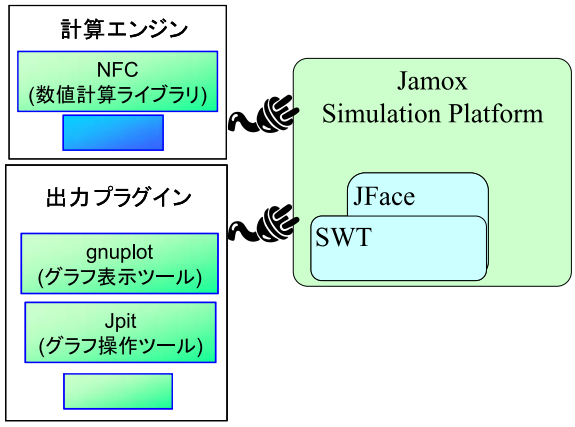


図 3.2 シミュレーションプラットフォーム

レーションを行うためにデフォルトでは数値計算ライブラリ NFC[7] を用いている。この計算エンジンは Jamox が提供する計算エンジン用のインターフェースを実装しているものを追加 (plugin) することが可能である。この機構を用いることで計算速度よりも計算精度を重視したライブラリを実装すれば精度を要求されるシミュレーションを行うことができるようになる。

また，出力プラグインもデフォルトでは本研究室内で開発中である Jpit[8] と gnuplot を使用するようになっているがこれも追加可能である。

3.4 シミュレーションアルゴリズム

Jamox のシミュレーションは，常微分方程式郡の数値積分により行う。数値積分の手法として

- Euler 法
- 改良 Euler 法
- Runge-Kutta-Fehlberg 法
- 4 次の Runge-Kutta 法

を使用することができる。*DifferentialEquation* インターフェースで定義した常微分方程式を解くソルバを *OdeSolver* クラスを継承して作成することで簡単に新しいアルゴリズムを追加することが可能である。

3.4.1 隣接文字列行列表現

Jamox ではシステム同士の結合関係を隣接文字列行列という形で保持している。**図 3.3** に示したブロック線図のように，各信号にユニークなノード番号を振ってモデルを管理している。隣接文字列行列では入力ノードを行番号，出力ノードを列番号に対応させることで表現している。点線で補った入力ポートと出力ポートは，隣接文字列行列を作成する際に必要となるため表示してある。

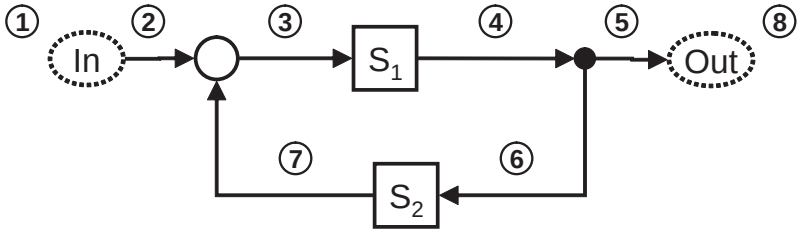


図 3.3 制御系のブロック線図

図 3.3 を表現する隣接文字列行列は次のようになる。

$$
\begin{array}{c}
\begin{array}{cccccccc} ① & ② & ③ & ④ & ⑤ & ⑥ & ⑦ & ⑧ \end{array} \\
\left[\begin{array}{c|c|c|c|c|c|c|c}
 & I & & & & & & \\ \hline
 & & P & & & & & \\ \hline
 & & & G_1 & & & & \\ \hline
 & & & & P & P & & \\ \hline
 & & & & & & & O \\ \hline
 & & & & & & G_2 & \\ \hline
 & & N & & & & & \\ \hline
 & & & & & & &
\end{array}\right]
\begin{array}{l} ① \\ ② \\ ③ \\ ④ \\ ⑤ \\ ⑥ \\ ⑦ \\ ⑧ \end{array}
\end{array}
\tag{3.1}
$$

(3.1) 式で**図 3.3** の接続関係を完全に表現できている。I は入力ポート，O は出力ポート，P は入力をそのまま出力，N は入力の符号を反転させて出力，G_i はシステムを表現している。

この隣接文字列行列を利用することで，多様で複雑なシステムを統一的な方法で表現できる。隣接文字列行列は文字列をキー，実際の制御システムを値としてデータを保持している。文字列として扱うことで，データ量が少なく，コンピュータ上で操作しやすいデータ構造である。

この隣接文字列行列から隣接行列 [16] を作成する。

3.4.2 隣接行列表現

隣接行列は，隣接文字列行列とは違い，文字列ではなく重みとして実際の値が入っている。実際に，シミュレーション等の計算を行うためには，隣接文字列行列から隣接行列を作成する必要がある。(3.1) 式の隣接文字列行列から作成した隣接行列は (**??**) 式のようになる。

①	②	③	④	⑤	⑥	⑦	⑧	
	$\underline{S}$							①
		I						②
			S_1					③
				I	I			④
							$\overline{S}$	⑤
						S_2		⑥
		$-I$						⑦
								⑧

(3.2)

I は入力をそのまま出力するオペレータ，$-I$ は入力の符号を反転して出力するオペレータ，S_i はシステムを表現するオペレータ，特に $\underline{S}$ は入力システム，$\overline{S}$ は出力システムを表す。

この隣接行列から，要素の代入やシステムの縮約，要素のソートなどの処理を行い，モデルの検証やシミュレーションを行っている。

第 4 章

Jamox の機能

4.1　パッケージ構成

Jamox は MVC(Model-View-Controller) アーキテクチャを採用している。

4.1.1　MVC アーキテクチャ

MVC アーキテクチャは，GUI アプリケーションの開発に広く用いられている手法で，アプリケーションをモデルとビュー，およびコントローラの三つの領域に分割して考える。つまり，何らかのデータを持つオブジェクト (モデル) をグラフィカルに描画し (ビュー)，編集する (コントローラ) ものになる。以下に MVC の概略と図 (**図 4.1**) を示す。

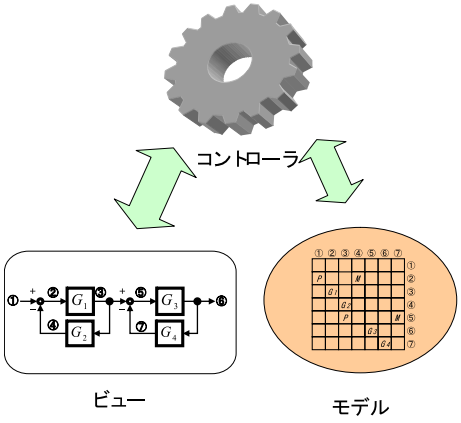


図 4.1 MVC アーキテクチャ

- モデル

  モデルは，表示されるデータを管理するものである。モデルは，ビューやコントローラについての情報を何も持っていない。

- ビュー

  ビューは，モデルの表示を受け持つ。ビューは，モデルの情報を取得し，その情報を元に描画を行うが，モデルに変更を加えることは一切できない。また，ビューはコン

トローラについての情報を持たないが，モデルに変更があった場合は，コントローラからモデルの変更を通知するメッセージが送られてくる。ビューは，このメッセージによって自身を更新する。

- コントローラ

  コントローラは，モデルを操作するためのユーザからの入力を受け付ける。コントローラは，モデルの変更を監視し，ユーザの操作などによって，モデルに変更が加えられた場合はモデルを設定し，ビューに通知する。

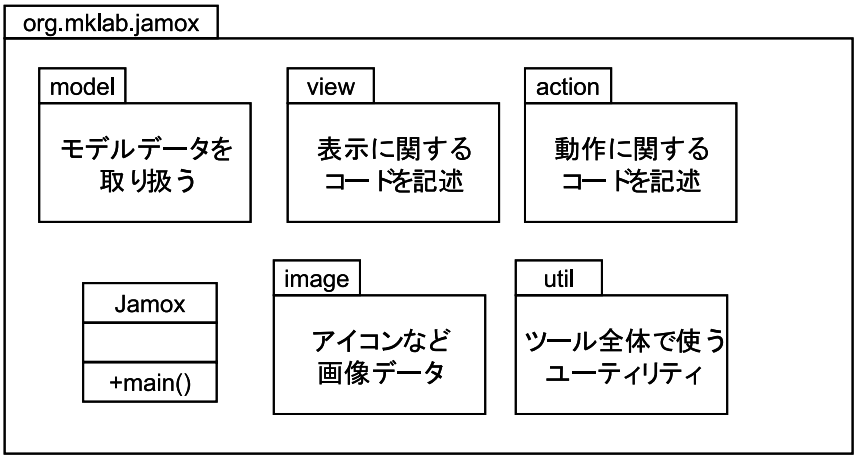


図 4.2 Jamox のパッケージ構成

コントローラは，モデルとビューの橋渡しをするものである。コントローラはモデルをそのビューにマッピングすることと，モデルに変更を加えることの両方の役割を担っている。

コントローラは直接モデルを変更するわけではなく，他にカプセル化されたコマンドがモデルの変更などを行う。

この MVC アーキテクチャに基づいて本ツールの作成を行っている。org.mklab.jamox パッケージの構成を**図 4.2** に示した。以下に各パッケージ及びクラスの説明を示す。

- org.mklab.jamox パッケージ

  メインクラスである Jamox クラスを格納するパッケージである。
- org.mklab.jamox.action パッケージ

  MVC モデルのコントローラに対応するパッケージである。このパッケージに属するクラスが主にモデルに対して操作を行うことになる。以下に各クラスの役割を示す。
- org.mklab.jamox.model パッケージ

  MVC モデルのモデルに対応するパッケージである。
- org.mklab.jamox.util パッケージ

  Jamox 全体で使うユーティリティを格納するパッケージである。

- org.mklab.jamox.image パッケージ

  画像を格納しておくパッケージである。
- org.mklab.jamox.view パッケージ

  MVC モデルのビューに対応するパッケージである。

4.2 Jamox の機能

Jamox の機能は以下の通りである[1]。

- ブロックライブラリの表示
- 簡単なマウス操作によるブロック線図の作成
- 作成したブロック線図のセーブ·ロード
- ブロックのパラメータ設定
- MATX エンジン [17] を利用した変数登録
- 登録された変数の一覧を表示
- ブロック線図のサブシステム化
- ブロックのカット・コピー・ペースト
- システムの妥当性検証
- 伝達関数演算
- 時間応答シミュレーション
- 周波数応答シミュレーション

4.2.1 ブロックライブラリの表示

図 4.3 に Jamox の画面構成を示す。メニューバー，ツールバー，ブロックを表すライブラリツリー，ブロック線図を作成する描画キャンバス，その切り替えのタブ，変数の登録などができるコンソールがある。ライブラリツリーは下記の XML で記述されたファイルに従って作成する。

[1] 2006 年 2 月現在

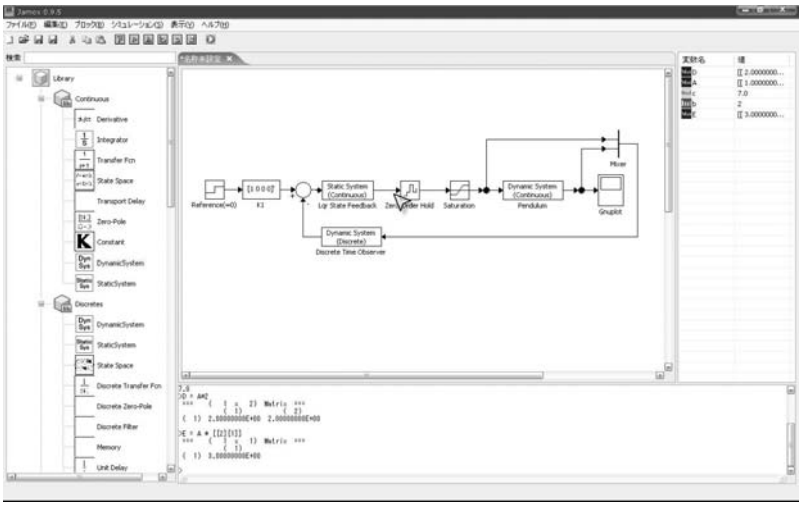


図 4.3 Jamox の画面構成

ブロックライブラリの XML

```
<?xml version="1.0" encoding="UTF-8"?>
<blockLibrary>
  <group name="Continuous">
    <block name="Integrator">
      <image src="image/continuous/integrat.gif"/>
      <className>
        org.mklab.jamox.model.block.continuous.BLIntegrator
      </className>
    </block>
    …中略…
    <block name="State Space">
      <image src="image/continuous/statespa.gif"/>
      <className>
        org.mklab.jamox.model.block.continuous.BLStateSpace
      </className>
    </block>
    <block name="Constant">
      <image src="image/continuous/constant.gif"/>
      <className>
        org.mklab.jamox.model.block.continuous.BLConstant
      </className>
    </block>
  </group>
  <group name="Discontinuous">
  …中略…
  </group>
</blockLibrary>
```

<group>要素で各ブロック要素をグループ化することができるようになっている。各ブロックの表示には<image>要素で指定されたイメージファイルで表示される。ない指定がない場合にはデフォルトイメージを使用するようにしている。実際にブロックをインスタンス化する場合には<className>要素で指定されたクラスが使用される。

図 4.4 に org.mklab.jamox.model.block パッケージのクラス図を示す。ライブラリの<className>

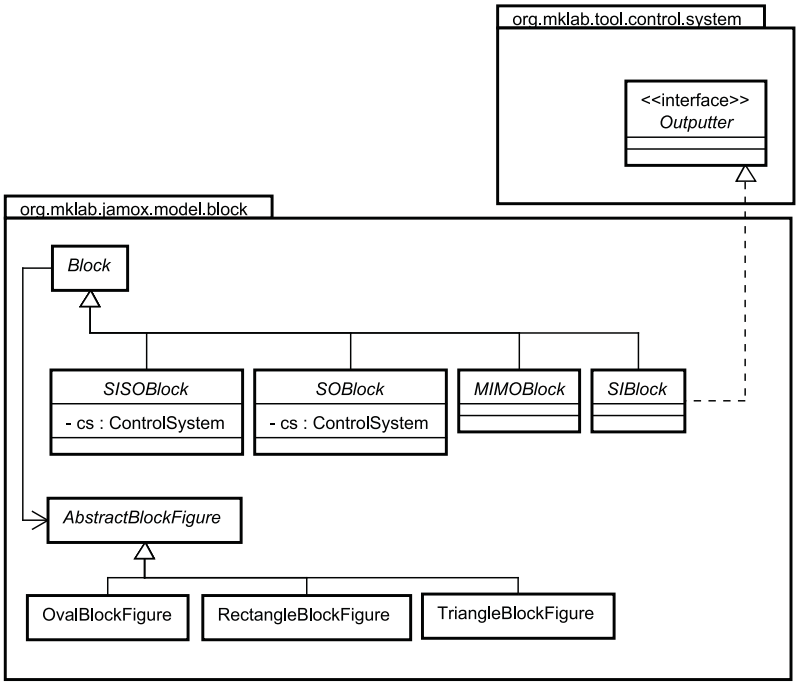


図 4.4 ブロックモデルのクラス図

要素で指定しているクラスは *Block* クラスのサブクラスである *SISOBlock*，*SIBlock*，*SOBlock*，*MIMOBlock* を継承したクラスである。

- *SISOBlock* クラス

  入力ポート，出力ポート共に 1 つのブロックである。伝達関数ブロック，状態空間表現ブロックなどがこれに該当する。
- *SIBlock* クラス

  出力ポートが 1 つだけ存在するブロック，つまり入力ブロックであるステップ信号ブロック，ランプ信号ブロックなどがこれに該当する。
- *SOBlock* クラス

  入力ポートが 1 つだけ存在するブロック，つまり出力ブロックである。Gnuplot ブロックや Jpit ブロックがこれに該当する。
- *MIMOBlock* クラス

  入力ポート，出力ポートのどちらかが複数存在するブロックである。加算器，マルチプレクサ，ディバイダ，分岐点，サブシステムなどがこれに該当する。

ここでいうポート数というのはシステムに入力，あるいは出力されるポートの数で，ポートの中の信号の数は複数であっても問題ないことに注意する。

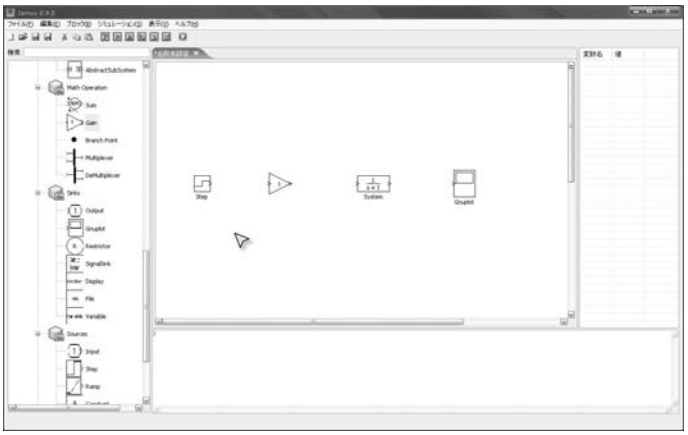


図 4.5 ブロックの配置

4.2.2 簡単なマウス操作によるブロック線図の作成

図 4.5 にライブラリツリーからモデリング領域にブロックを配置したときのスクリーンショットを示す。この配置はブロックライブラリから配置したいブロックをモデリング領域にドラッグ&ドロップすることで行うことができる。

ドラッグ&ドロップを実現するために SWT の dnd パッケージ (org.eclipse.swt.dnd.*) を利用している。ドラッグ&ドロップの概念図を**図 4.6** に示す。

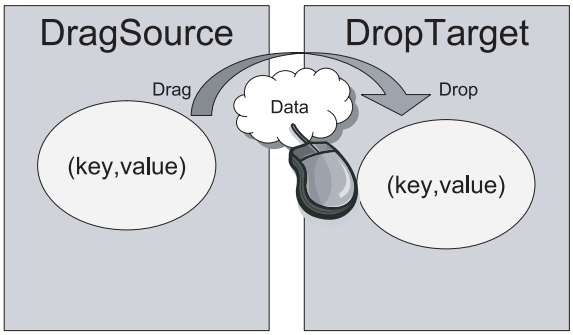


図 4.6 ドラッグ&ドロップの概要

ドラッグする元となるデータを持つものを DragSource，そのデータをドロップする場所を DropTarget と呼ぶ。ドラッグしたときに何をデータとして渡すのかというものは DragSourceEvent の data フィールドに設定する。ここで渡すデータは前述の XML データで記述されいてるクラス名を渡している。ドラッグを可能にするためにはドラッグの元になるウィジェット addDragSourceListener() メソッドを用いてリスナーを追加することで可能である。

DragSource からデータをドロップしてくるクラスには DropListener を追加しなければいけない。ドロップ側にイベントが起きたときに呼ばれるのはドラッグしてウィジェットの中に入ってきた dragEnter() メソッドとマウスを放したときに起きる drop() メソッドである。

ドロップされたデータは DropTargetEvent の data フィールドに保持されている。ここで送られてくるのはドラッグを開始したときに設定したクラス名である。このクラスを元にモデルのデータ及びブロックの内部構造を生成している。

ブロックの描画では，AbstractBlockFigrue クラスのサブクラスの draw() メソッドを利用して，ブロックを描いている。

ブロックの移動

一度キャンバス内に置いたブロックはマウスでドラッグすることで自由に動かすことができる。このときキーボードの矢印キーを押しても動かすことが可能である。Alt キーを押しながら矢印キーを押すことで細かい移動を行うことができる。

ブロック同士の接続

ブロックからブロックへの接続を行う方法を以下に示す。あるブロックの出力ポートから違うブロックの入力ポートまでドラッグ&ドロップすることでブロック同士のリンクを作成することができる。ドラッグしている間，ドラッグ元のポートから線が伸び，接続し終えるとポートを示すアイコンは消滅し，線は信号の向きを示す矢印に変わる。

この間の線は信号線を表現しており，直感的に操作できるように工夫してある。また Ctrl キーを押した状態で次のブロックをクリックすることでもブロック同士のリンクを作成することができる。この操作は直感的にはわかりやすいが素早くブロック同士をリンクすることができるようになっている。ブロック同士の結合をした図を**図 4.7** に示す。

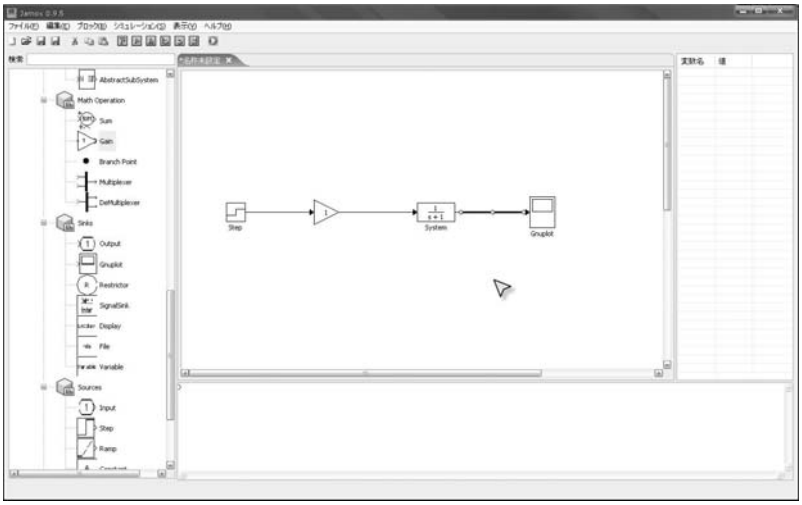
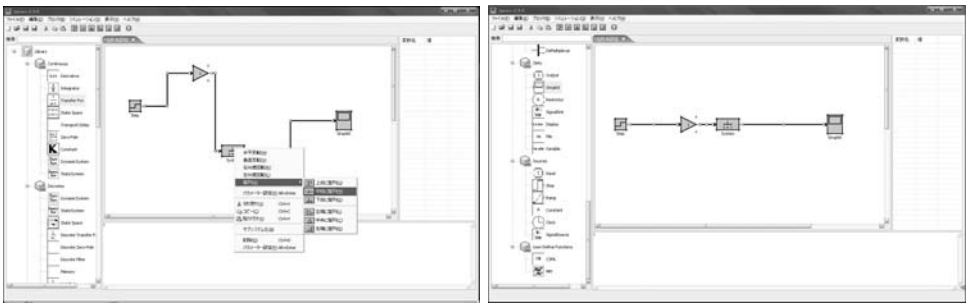


図 4.7 ブロック同士の結合



(a) 右クリックメニュー (中段揃え)　　(b) 中段に整列後

図 4.8 ブロックを中段に整列

4.2.3 ブロックの整列

作成したブロック線図は簡単に綺麗に整列させることができる。整列してブロックを描くことでブロック線図がよりわかりやすくなるという利点がある。**図 4.8(a)** に適当に並べられたブロックのうち整列したい複数ブロックを選択した状態でで右クリックメニューを出した図を示した。

ここで「整列」「中段に整列」と順に選択することで，縦方向に中央にブロックを整列させることができる。整列後のブロック線図を**図 4.8(b)** に示す。

4.2.4 ブロックのパラメータ設定

図 4.9(a) に示すように各ブロックをダブルクリックするか，ブロックを右クリックし「パラメータ設定」メニューを選択することで**図 4.9(b)** に示すパラメータ設定画面を表示することができる。図は伝達関数のパラメータ設定画面である。

4.2.5 MATX エンジンを利用した変数登録

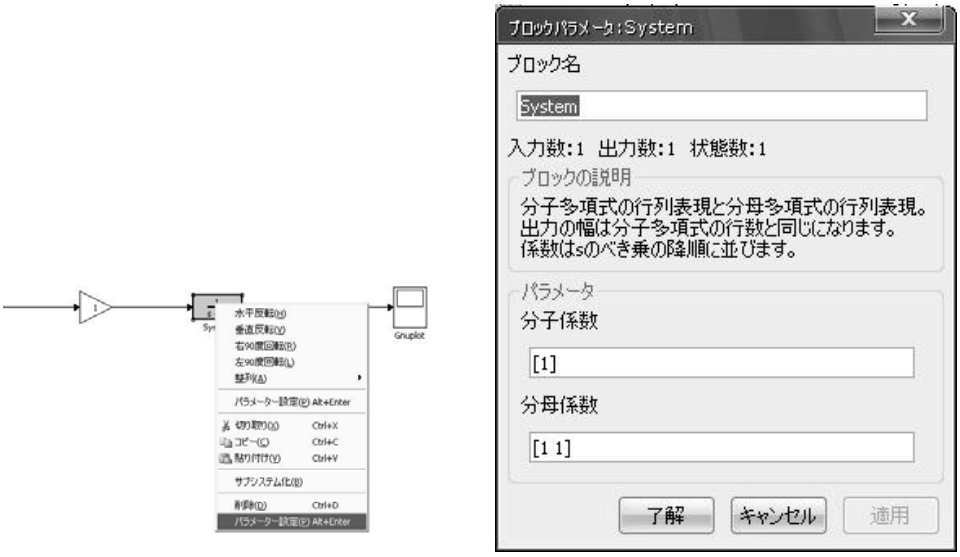
コンソールに打ったコマンドを MATX エンジンに送って変数として登録し，その変数を上記のパラメータ設定で利用することが可能である。

例えば，コンソールに，

$$A = [1 \;\; 1]$$

と入力しエンターキーを押すと，それが変数 A として登録され，さまざまな箇所で利用可能となる。

その他にもコンソールを利用することで，行列の固有値などを計算することも可能である。



(a) ブロックの右クリックメニュー (パラメータ設定)

(b) パラメータ設定画面 (TransferFcn)

図 4.9 ブロックパラメータの設定

4.2.6 登録された変数の一覧を表示

上記の方法で登録した変数の一覧が表形式で表示される。また，変数の値をここで変更することも可能である。変数テーブル画面を**図 4.10** に示す。

図 4.10 変数テーブル

4.2.7 ブロック線図のセーブ・ロード

作成したブロック線図は，XML 形式の jamox ファイルとして保存することができる。

以下のファイルはブロック線図を保存したときのファイルである。設定したパラメータ情報をブロックの要素の<param>で保存してある。保存されているパラメータはパラメータ設定ダイアログで入力される値そのものである。また，このファイルを読み込んでブロック線図を復元することが可能になっている。この XML の扱いには，JAXB によるデータバインディングの手法を利用している。[付録 B]

あるブロック線図の jamox ファイル (sample.jamox)

```
<?xml version="1.0" encoding="UTF-8"?>
<bdml version="0.9.2">
  <system name="sample.jamox">
    <block name="-F" id="123">
      <port id="124" type="input">
        <figure x="274" y="243" height="6" width="6" angle="180">
          <class>org.mklab.jamox.model.PortFigure</class>
        </figure>
      </port>
      <port id="125" type="output">
        <figure x="214" y="243" height="6" width="6" angle="180">
          <class>org.mklab.jamox.model.PortFigure</class>
        </figure>
      </port>
      <param name="Element">
        [3.16227766E+02  5.19327117E+02
        2.07476867E+02 5.81519066E+01]
      </param>
      <figure x="214" y="228" height="30" width="60" angle="0"
          horizontalReflect="true" verticalReflect="false">
        <class>org.mklab.jamox.model.BLConstantFigure</class>
        <name>-F</name>
      </figure>
      <class>org.mklab.jamox.model.BLConstant</class>
    </block>
  </system>
</bdml>
```

4.2.8 ブロックのカット・コピー・ペースト

図 4.11 に示すようにブロックを選択し，右クリックメニューから「コピー」もしくは「カット」を選択するか「編集」メニューから「コピー」もしくは「カット」を選択することでブロックの情報をクリップボードに格納することができる。クリップボードに保存することができるのはテキスト形式または byte[] 形式なので ByteArray 化を行う必要がある。

本ツールでは，コピーもしくはカットを次のような手法で行っている。選択されたブロックの情報を .jamox ファイル同様のフォーマットでクリップボードに格納する。ただし，ブロック同士の接続 (BlockLink) は，接続元のブロックと接続先のブロックの両方が選択されている場合に限り BlockLink の情報もクリップボードに格納する。

ブロックのペーストを行う際は，コピーしたブロック線図の情報を .jamox 形式の XML から復元を行う。その際，コピー元のブロック線図とコピー先のブロック線図のインスタンスは複製されたものになるようにしている。ブロックの接続も同様に複製されたブロック同士がリンクされるようにしている。そして，ブロックがペーストされたあとペーストされた

ブロックを選択状態にする。これは非常に直感的である。

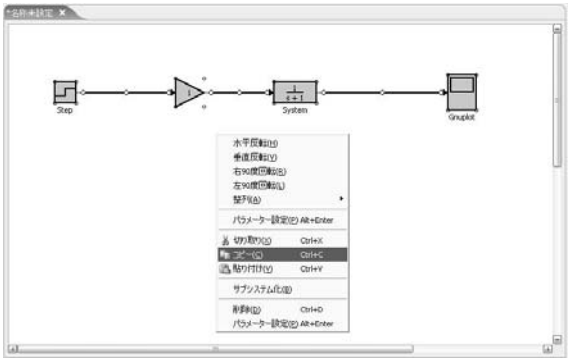
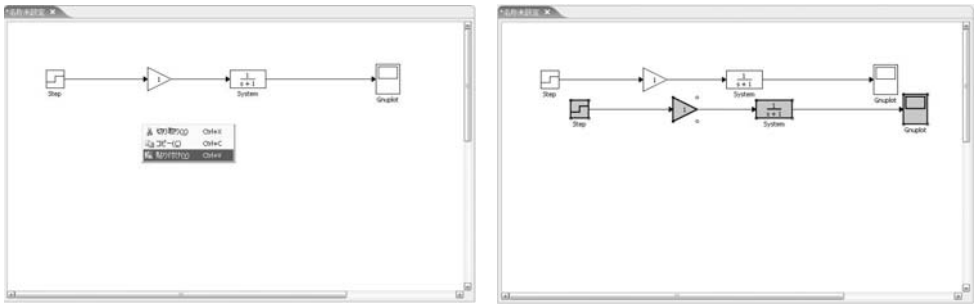


図 4.11 ブロックの右クリックメニュー



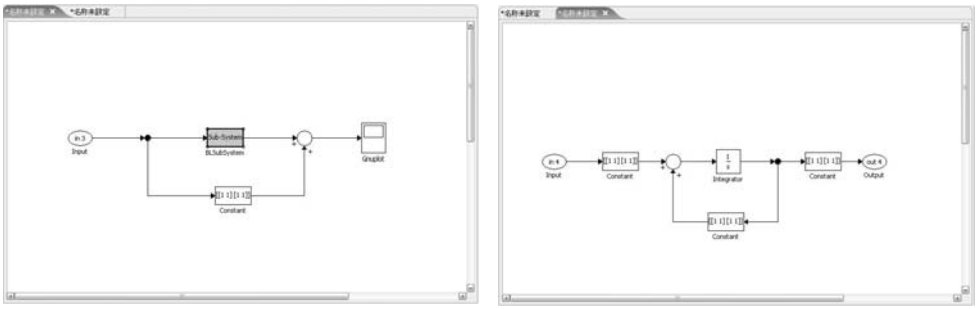
(a) 右クリックメニュー　　(b) ペースト後の画面

図 4.12 コピーしたブロックのペースト

4.2.9 ブロック線図のサブシステム化

ブロック線図で複雑なシステムを設計していく場合，ブロック数が増えすぎてわかりにくくなる場合がある。こういうときにあるシステムのまとまりをサブシステムとして定義するとわかりやすいブロック線図が作成できる。また，作成したサブシステムを一つのブロックとして取り扱うことができるので再利用性に優れる。

図 4.13 にサブシステムの利用画面を示す。左の**図 4.13(a)** のサブシステムのブロックをダブルクリックすると，サブシステムの構成要素を別のタブで開くことができる。そのサブシステムの中身を**図 4.13(b)** に示す。



(a) root システム　　(b) サブシステムの中身

図 4.13 コピーしたブロックのペースト

サブシステムの中にサブシステムを含むシステムを構築することも可能である。また，現在の仕様ではサブシステムには必ず IN ブロックと OUT ブロックがなければならない。

サブシステムの作り方には二通りある。一つめはサブシステムブロックを新規に配置する方法，もう一つはサブシステムにしたいブロックを選択して「サブシステム化」ボタンをクリックする方法である。

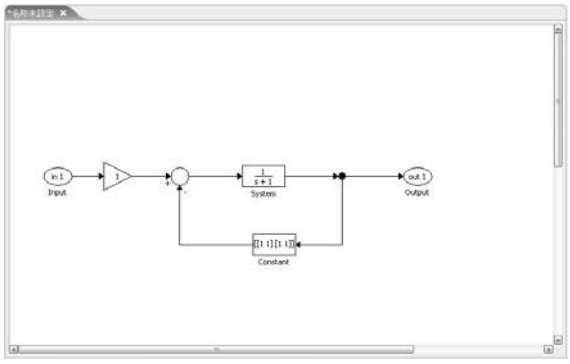


図 4.14 サブシステム化前のシステム

図 4.14 にサブシステム化を行う前のブロック線図 (倒立振子制御システム) を示した。システムの中でサブシステムにしたいブロックを選択し右クリックメニューから「サブシステム化」を選択する。その場合の図を**図??**に示す。

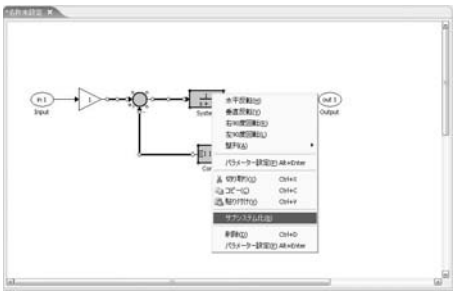
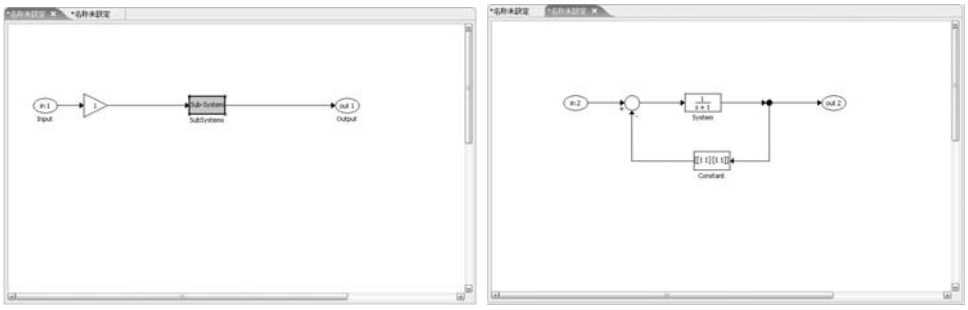


図 4.15 サブシステム化メニュー



(a) サブシステム化されたシステム　　(b) サブシステムの中味

図 4.16 サブシステム化

図 4.16(a) に示すように，選択されたブロックは「サブシステム」に置き換わる。そのサブシステムをダブルクリックすることでサブシステムの中味を確認することができる。サブシステムの中身は**図 4.16(b)** に示す。

シミュレーションはシステム全体に対して行われることに注意する。

4.2.10 システムの妥当性検証

作成したモデルが巨大であったり複雑である場合，システムの設計以前にモデリングで間違ってしまう場合が出てくる。シミュレーション中にエラーが出た場合，どこが間違っているのかを探し出すのに苦労することがある。Jamox を利用すると，作成したモデルが妥当であるかどうかをシミュレーションを行う前に解析・検証することができる。ここで利用しているアルゴリズムは**??**で述べた直達項ループの発見法と，接続行列利用による解析である。

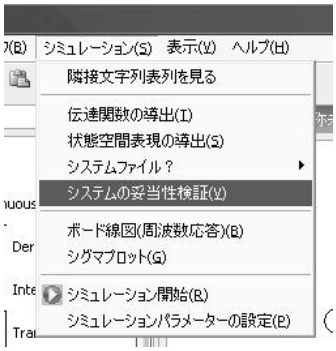
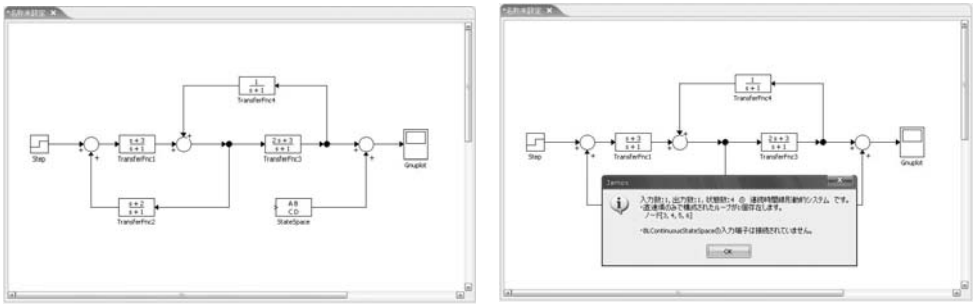


図 4.17 妥当性検証メニュー

図 4.17 に妥当性検証メニューを示した。このボタンをクリックすると，現在アクティブになっているモデルに対して妥当性の検証を行う。システムが妥当である場合にはシステムがどのような性質を持っているかを表示する。

システムの情報表示

図 4.18(a) に直達項のみで構成されるループを含むシステムを示す。このシステムは図中の TransferFcn1 ブロック，TransferFcn2 ブロック，TransferFcn3 は伝達関数の分子と分母の次数が等しい，つまり直達項を持つシステムである。TransferFcn4 はプロパーなため直達項を持たない。



(a) 直達項のみのループを含むシステム　　(b) システム情報解析結果

図 4.18 システム情報解析結果

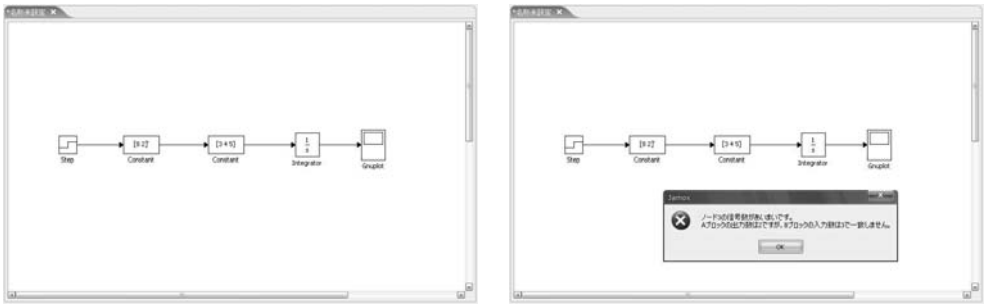
このシステムに対して**図 4.17** に示した妥当性検証メニューを実行すると**図 4.18(b)** のような結果を得る。このシステムは 1 入力 1 出力 4 状態の連続時間線形動的システムであることがわかる。直達項のみで構成されたループが 1 個存在し，そのノード番号は 3，4，5，6

から構成されていることを示している。また，BLContinuousStateSpace の入力ポートに何も接続していないため何も接続されていない警告が出ている。

信号数の違いによるエラーの発見

あるシステムの入力が m 次元であるのに対し，その出力先のシステムの次元が m 以外である場合，システムが妥当でないことを意味する。

図 4.19(a) に出力の次元と入力の次元が等しくないシステムを示す。このシステムに対し妥当性の検証を行うと**図 4.19(b)** の結果を得る。メッセージからノード 3 の信号数が 2 と 3 で等しくないことがわかる。開発者はノード 3 のあたりを重点的にチェックすることでシステムを妥当なものにすることができる。



(a) 出力信号数と入力信号数が一致しないシステム

(b) モデルエラーの表示画面

図 4.19 モデルエラーの解析結果

4.2.11 伝達関数演算

図 4.20 に 1 型 2 次系サーボシステムを示す。

Jamox では，線形システムに対して伝達関数の演算を行うことができる。入力から出力までの一巡伝達関数を求めるには，シミュレーションメニューから「伝達関数の導出」をクリックすればよい。入力ブロックと出力ブロックが複数ある場合には，伝達関数行列を求めることになる。

図 4.21 に伝達関数の取得について示す。**図 4.21(a)** を選択すると，システムの伝達関数を求め，コンソールに出力する。また，その得られた結果を**図 4.21(b)** に示す。システムが線形でない場合にはエラーメッセージが表示される。

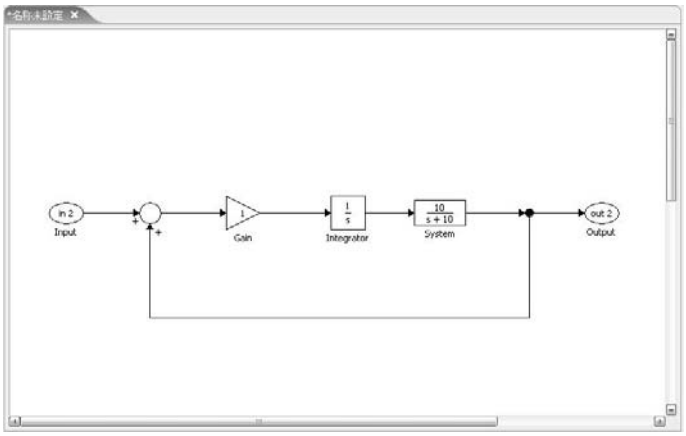
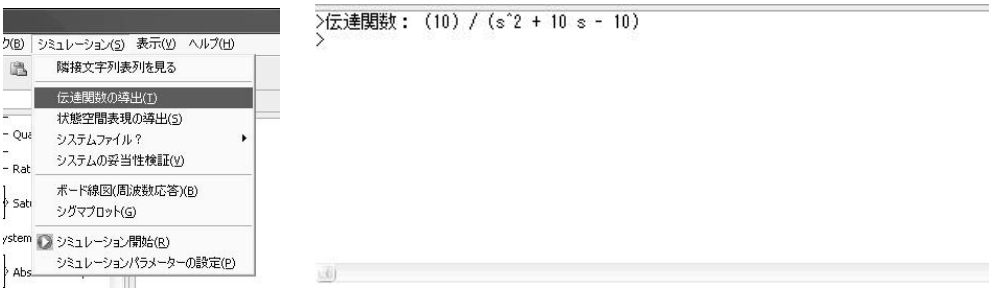


図 4.20 1 型 2 次系サーボ



(a) 伝達関数メニュー　　(b) 取得された伝達関数

図 4.21 伝達関数の取得

4.2.12 周波数応答シミュレーション

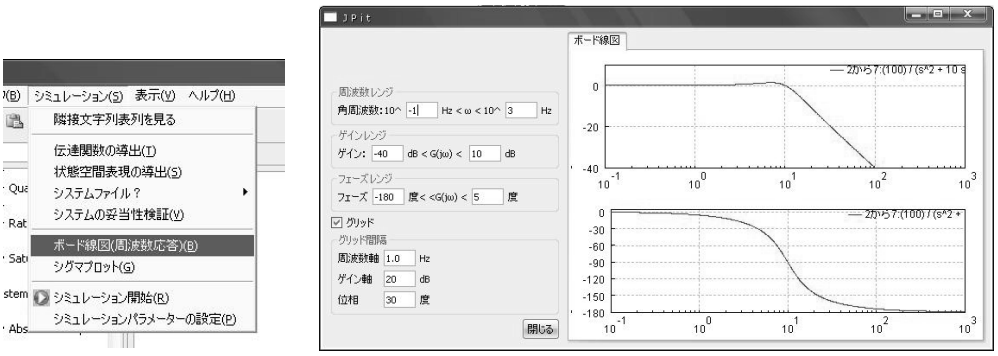
Jamox では周波数応答シミュレーションを行うことができる。その様子を**図 4.22** に示す。**図 4.22(a)** に示す，ボード線図メニューを実行すると，別ウィンドウにボード線図が表示される。**図 4.22(b)** は**図 4.20** のボード線図の結果である。

4.2.13 時間応答シミュレーション

Jamox では 3.4 で述べたアルゴリズムで時間応答シミュレーションを行うことができる。**図 4.23** に Jamox で作成した倒立振子制御系のモデルを示す。

シミュレーションを行うには，シミュレーション時間，ソルバを選択する必要がある。シミュレーションパラメータの設定画面を**図 4.24(a)** に示す。

ソルバは *OdeSolver* インターフェースを用意したものを作成すれば自動的に追加されるようになっている。



(a) ボード線図メニュー (b) ボード線図

図 4.22 Jpit によるボード線図の表示

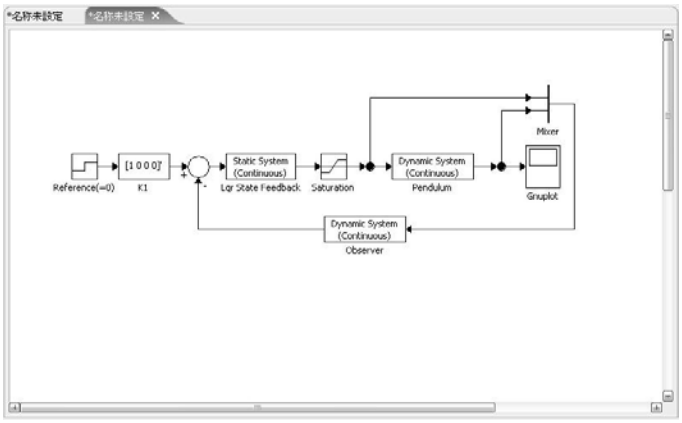


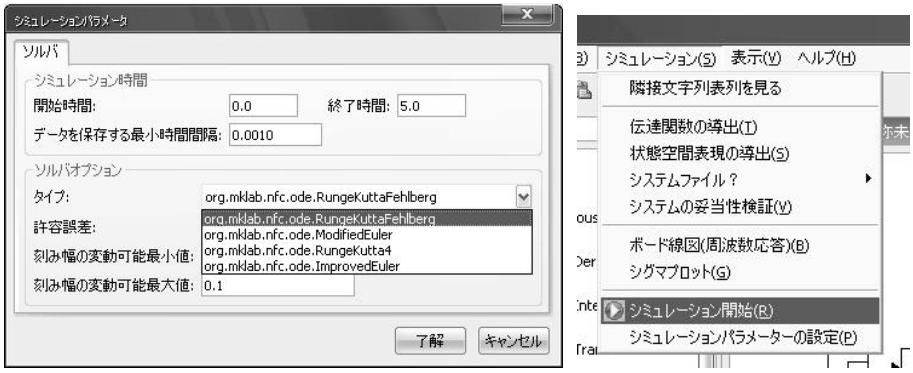
図 4.23 倒立振子系モデル

シミュレーションパラメータの設定が完了したらあとはメニューから「シミュレーション開始」をクリック (**図 4.24(b)**) するとシミュレーションが行われ，出力ブロックに結果が渡される。

出力ブロックにデータが渡され，処理が行われる。Gnuplot ブロックにデータが渡された場合は，Gnuplot を用いて出力を描画する。倒立振子制御系のシミュレーション結果を**図??**に示す。

4.2.14 ユーザー定義ブロックの作成

Jamox では，ブロックライブラリのブロックを結合したシステムを作成するだけでなく，Java クラスを記述することで線形，非線形などシステムの形態を問わず自由にシステムを作成することが可能である。ステップ信号入力を例に，実際にどんなクラスを作成すれば良



(a) シミュレーションパラメータの設定　　(b) シミュレーション実行メニュー

図 4.24 シミュレーションパラメータの設定画面と実行方法

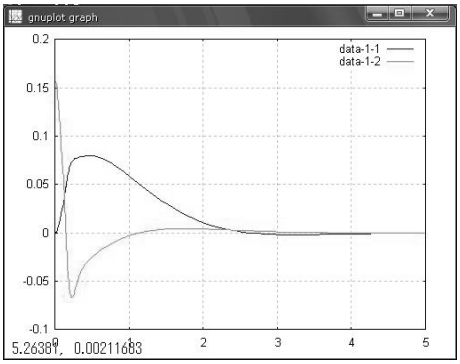


図 4.25 シミュレーション結果の表示

いかを説明する。

図 4.4 のように *SISOBlock* と *SOBlock* は ControlSystem をフィールドに持っている。

まず，システムの特性に合わせたクラスを継承して *SystemOperator* を作成する。ステップ信号入力の場合は連続時間静的システムなので *AbstractContinuousStaticSystem* クラスを継承して次のように記述する。

StepSystem.java

```
public class StepSystem extends AbstractContinuousStaticSystem {
  @Parameter(name="initialValue", description="初期値")
  private double value;
  @Parameter(name ="finalValue", description="最終値")
  private double initialValue;
  @Parameter(name="delayTime", unit=SIunit.s, description="遅れ時間")
  private double delay;
  /**
   * コンストラクター
   * @param val ステップ値
   * @param initialValue 初期値
   * @param delay ステップ開始時刻
   */
  public StepSystem(double val, double initialValue, double delay) {
    super(0, 1);
    setHasDirectPath(false);// 直達項なし
    setInputRequired(false);// 入力なし
    this.value = val;
    this.initialValue = initialValue;
    this.delay = delay;
  }
  /**
   * @param t
   *        時間
   * @return 時刻 t における出力の行列
   */
  @Override
  public Matrix outputEq(double t) {
    if (t < delay) {
      return new RealMatrix(new double[] {initialValue});
    }
    return new RealMatrix(new double[] {value});
  }
}
```

次に，Jamox と関連付けるためのモデルである *Block* クラスのサブクラスを作成する。ステップ信号ブロックは入力ブロック，つまり *SOBlock* クラスを派生させる必要がある。作成する必要があるのはデフォルトパラメータを設定する setDefaultParameter() メソッドと GUI から入力されたパラメタを適用する apply() メソッドである。setDefaultParameter() 内で，各パラメータに対して入力の値の種類 (テキスト，チェックボックス，ファイル名等) と制約 (行列のみ許可等) を付加することができる。

BLStep.java

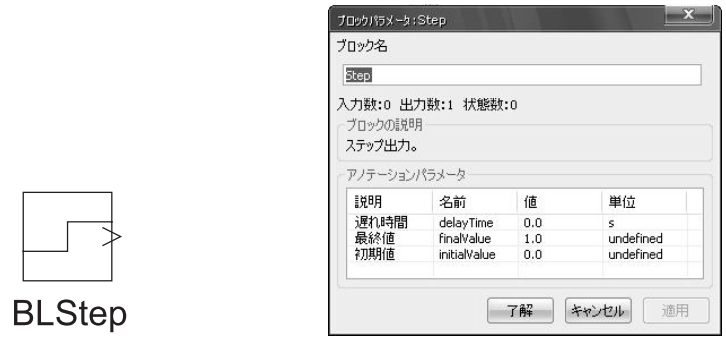
```
public class BLStep extends SOBlock {
  //ステップシステム
  private StepSystem stepSystem;
  /**
   * コンストラクター
   */
  public BLStep() {
    super();
    setHintsOfBlock("ステップ出力。"); //ブロックの説明
    setName("Step"); //ブロックの名前
  }
  @Override
  protected void setDefaultParameter() throws ParameterFormatException {
    stepSystem = new StepSystem();
  }
  /**
   * @throws ParameterFormatException 望ましくない値が入力されたときに発生
する例外
   */
  @Override
  public void apply() throws ParameterFormatException {
    setControlSystem(new ControlSystem(stepSystem));
  }
}
```

最後にブロックを表示させる際にどのようなブロックにするかを決定するクラスを作成する。デフォルトではブロックで作成したクラス名のポストフィックスが”Figure” のクラスを使用する。存在しない場合は RectangleBlockFigure が使用されるので，必ず作成しなければならないクラスではない。

以下にステップ入力信号を描くクラスを示す。コンストラクタでサイズを設定しているほか，drawBlock() メソッドでブロックの描画を行っている。ブロックに設定してあるパラメタを読み取ることも可能である。

BLStepFigure.java

```
public class BLStepFigure extends RectangleBlockFigure {
  /**
   * @param x x 座標
   * @param y y 座標
   * @param block ブロック要素
   */
  public BLStepFigure(final int x, final int y, final Block block) {
    super(x, y, block);
    setSize(30, 30);
  }
  /**
   * ステップブロックを表す図を描く
   * @param gc グラフィックコンテキスト
   */
  @Override
  public void drawBlock(final GraphicsAdapter gc) {
    //ブロックの描画をする
    super.drawBlock(gc);
    //ステップを描く
    int[] apecPoint = {x, y + width * 2 / 3, x + width / 2, y + width *
    2 / 3, x + width / 2, y + width / 3, x + width, y + width / 3};
    gc.drawPolyline(apecPoint);
  }
}
```



(a) 描画されたブロック (b) ブロックパラメータ設定画面

図 4.26 作成したステップブロック

4.2.15 MATX 形式 (mm ファイル) から Java ファイルへの変換

Jamox では MATX 形式のファイルをブロックとして読み込むことができる MM ブロックを提供している。このブロックを利用することで数値計算言語 MATX 文法を利用して作成

したシステムをブロックとして再利用することができる。MM ブロックで利用可能な mm ファイルが満たすべき条件は以下の通り。

- `Func void init()` で初期化処理を行っている
- `Func Matrix state_eq(t, X, U)` で状態の計算を行っている
- `Func Matrix output_eq(t, X)` で出力の計算を行っている

この条件を満たした mm ファイルであれば，Jamox の MM ブロックに入力することで matj を利用し *SystemEquation* を継承した Java クラスを生成，コンパイル後，ControlSystem として登録する。

仕様を満たした mm ファイルの例として，倒立振子の非線形モデルを以下に示す。

倒立振子の非線形モデル (MM ファイル)(1/3))

```
Real m1, m2, J, l, g, Fr,Cr, a0, alpha;
Real a32,a33,a34,a35,a42,a43,a44,a45;
Real b3,b4,cc1,cc2;
/**
 * 初期設定
 */
Func void init() {
 Real alpha0;
 m1 = 0.16;
 m2 = 0.039;
 Fr = 2.6;
 Cr = 4.210e-4;
 l  = 0.121;
 J  = 4.485e-4;
 a0 = 0.1;
 g  = 9.8;
 alpha0 = (m1 + m2)*J + m1*m2*l*l;
 alpha = m2*m2*l*l/alpha0;
 a32 = -(m2*l)*(m2*l)*g;
 a33 = -Fr*(J + m2*l*l);
 a34 = m2*l*Cr;
 a35 = (J + m2*l*l)*m2*l;
 a42 = (m1 + m2)*m2*l*g;
 a43 = m2*l*Fr;
 a44 = -(m1 + m2)*Cr;
 a45 = -(m2*l)*(m2*l);
 b3  = (J + m2*l*l)*a0;
 b4  = -m2*l*a0;
 a32 = a32/alpha0;
 a33 = a33/alpha0;
 a34 = a34/alpha0;
 a35 = a35/alpha0;
 a42 = a42/alpha0;
 a43 = a43/alpha0;
 a44 = a44/alpha0;
 a45 = a45/alpha0;
 b3  = b3/alpha0;
 b4  = b4/alpha0;
 cc1 = 1.0;
 cc2 = 1.0;
}
```

倒立振子の非線形モデル (MM ファイル)(2/3))

```
/**
 * 状態方程式
 * @param t 時間
 * @param x 状態
 * @param u 入力
 * @return 状態の微分
 */
Func Matrix state_eq(t, x, u)
 Real t;
 Matrix x;
 Matrix u;
{
 Matrix dx;
 Real x2,x3,x4,u1;
 Real c2,s2,determ;

 x2 = x(2,1);
 x3 = x(3,1);
 x4 = x(4,1);
 u1 = u(1,1);

 c2 = cos(x2);
 s2 = sin(x2);
 determ = (1 + alpha*s2*s2);

 dx = Z(4,1);
 dx(1,1) = x3;
 dx(2,1) = x4;
 dx(3,1) = (a32*s2*c2 + a33*x3 + a34*c2*x4 + a35*s2*x4*x4 +
b3*u1)/determ;
 dx(4,1) = (a42*s2 + a43*c2*x3 + a44*x4 + a45*s2*c2*x4*x4 +
b4*c2*u1)/determ;
 return dx;
}
```

倒立振子の非線形モデル (MM ファイル)(3/3))

```
/**
 * 出力方程式
 * @param t 時間
 * @param x 状態
 * @return 出力
 */
Func Matrix output_eq(t, x)
 Real t;
 Matrix x;
{
 Matrix y;
 Real x1, x2;
 x1 = x(1,1);
 x2 = x(2,1);
 // レールの幅 (中央から 0.15[m]) を越えるとシミュレーションを停止
する
 if (abs(x1) > 0.16){
  OdeStop();
 }
 y = Z(2,1);
 y(1,1) = cc1*x1;
 y(2,1) = cc2*x2;
 return y;
}
```

このファイルを matj を用いて Java ファイルに変換したあと，*ControlSystem* として取り扱えるように *AbstractDynamicSystem* や *AbstractStaticSystem* を継承させる情報を織り込む。このクラスをコンパイルすることで一つの非線形モデルを記述した mm ファイルを一つのブロックとして読み込むことが可能になる。変換された Java ファイルを以下に示す。

倒立振子の非線形モデル (Java)(1/2))

```
import java.lang.Math;
import org.mklab.matj.util.MaTXFunction;
import org.mklab.nfc.Matrix;
import org.mklab.nfc.RealMatrix;
import org.mklab.tool.control.system.AbstractDynamicSystem;
public class pendulum1 extends AbstractDynamicSystem {
  private double m2;
  private double m1;
  private double a45;
  private double alpha;
  private double b4;
  private double a44;
  private double b3;
  private double a43;
  private double a42;
  private double Cr;
  private double cc2;
  private double Fr;
  private double cc1;
  private double J;
  private double l;
  private double g;
  private double a35;
  private double a34;
  private double a33;
  private double a32;
  private double a0;
  public pendulum1(int inputNumber, int outputNumber, int stateNumber) {
    super(inputNumber,outputNumber,stateNumber);
    setInputRequired(true);
    setHasDirectPath(true);
    double alpha0;
    m1 = 0.16;
    m2 = 0.039;
    Fr = 2.6;
    Cr = 4.21E-4;
    l = 0.121;
    J = 4.485E-4;
    a0 = 0.1;
    g = 9.8;
    alpha0 = (m1 + m2) * J + m1 * m2 * l * l;
    alpha = m2 * m2 * l * l / alpha0;
    a32 = -(m2 * l) * (m2 * l) * g;
    a33 = -Fr * (J + m2 * l * l);
    a34 = m2 * l * Cr;
    a35 = (J + m2 * l * l) * m2 * l;
    a42 = (m1 + m2) * m2 * l * g;
    a43 = m2 * l * Fr;
    a44 = -(m1 + m2) * Cr;
    a45 = -(m2 * l) * (m2 * l);
    b3 = (J + m2 * l * l) * a0;
    b4 = -m2 * l * a0;
    a32 = a32 / alpha0;
    a33 = a33 / alpha0;
    a34 = a34 / alpha0;
    a35 = a35 / alpha0;
    a42 = a42 / alpha0;
    a43 = a43 / alpha0;
    a44 = a44 / alpha0;
    a45 = a45 / alpha0;
    b3 = b3 / alpha0;
    b4 = b4 / alpha0;
    cc1 = 1.0;
    cc2 = 1.0;
  }
```

倒立振子の非線形モデル (Java)(2/2))

```
  public Matrix stateEq(double t, Matrix x, Matrix u) {
    Matrix dx;
    double x2;
    double x3;
    double x4;
    double u1;
    double c2;
    double s2;
    double determ;
    x2 = ((RealMatrix)x).getElement(2, 1);
    x3 = ((RealMatrix)x).getElement(3, 1);
    x4 = ((RealMatrix)x).getElement(4, 1);
    u1 = ((RealMatrix)u).getElement(1, 1);
    c2 = Math.cos(x2);
    s2 = Math.sin(x2);
    determ = 1 + alpha * s2 * s2;
    dx = MaTXFunction.Z(4, 1);
    ((RealMatrix)dx).setElement(1, 1, x3);
    ((RealMatrix)dx).setElement(2, 1, x4);
    ((RealMatrix)dx).setElement(3, 1, (a32 * s2 * c2 + a33 * x3 + a34 *
c2 * x4 + a35 * s2 * x4 * x4 + b3 * u1) / determ);
    ((RealMatrix)dx).setElement(4, 1, (a42 * s2 + a43 * c2 * x3 + a44 *
x4 + a45 * s2 * c2 * x4 * x4 + b4 * c2 * u1) / determ);
    return dx;
  }
  public Matrix outputEq(double t, Matrix x) {
    Matrix y;
    double x1;
    double x2;
    x1 = ((RealMatrix)x).getElement(1, 1);
    x2 = ((RealMatrix)x).getElement(2, 1);
    if (Math.abs(x1) > 0.16) {
      MaTXFunction.OdeStop();
    }
    y = MaTXFunction.Z(2, 1);
    ((RealMatrix)y).setElement(1, 1, cc1 * x1);
    ((RealMatrix)y).setElement(2, 1, cc2 * x2);
    return y;
  }
}
```

第5章

Jamoxを利用した制御系設計

5.1　古典制御問題

5.1.1　PID 制御

例題 [18, p151]
制御対象がつぎの伝達関数で与えられるとする。

$$P(s) = \frac{10}{(s+1)(s+10)}$$

まずはじめに，次のような比例ゲインによる P 補償を考える。

$$K_P = 10$$

このときの開ループ系の伝達関数を求めるためには本ツールを用いて 5.1 のようにモデルを作成すれば良い。このときにシミュレーションメニューから「ボード線図」を選ぶことで**図 5.3** のようなボード線図を得ることができる。

開ループゲインは簡単に計算できるが**図 5.3** を見てもわかるように $L_P(s) = 100/(s^2 + 11s + 10)$ である。この図で $|L_P(0)|$ は無限大でないので，定常位置偏差が残ることがわかる。

図 5.4 のように実際に閉ループ系を作成し，シミュレーションメニューから「ステップ応答」を選ぶことで**図 5.5** のステップ応答を得ることができる。

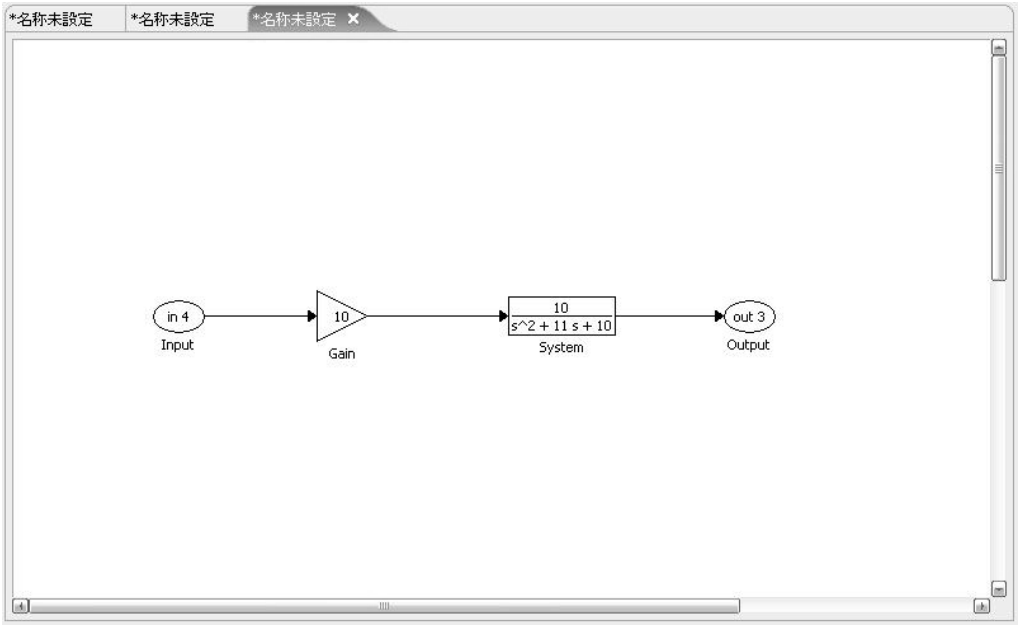


図 5.1 $K_P = 10$ のときの開ループゲインを求めるためのブロック線図

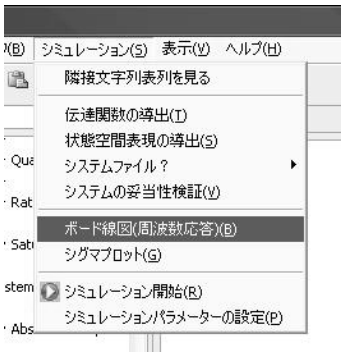


図 5.2 シミュレーションメニュー

図 5.5 を見て，たしかに定常位置偏差が残っていることがわかる。
そこで，定常位置偏差を取り除くために PI 補償を以下のようにする。

$$K_{PI}(s) = \frac{s+1}{s}$$

つまり，**図 5.6** のようなモデルを作成すれば良い。このときの開ループ系の伝達関数は $L_{PI}(s) = 10/(s(s+10))$ になる。この開ループゲインは**図 5.7** である。

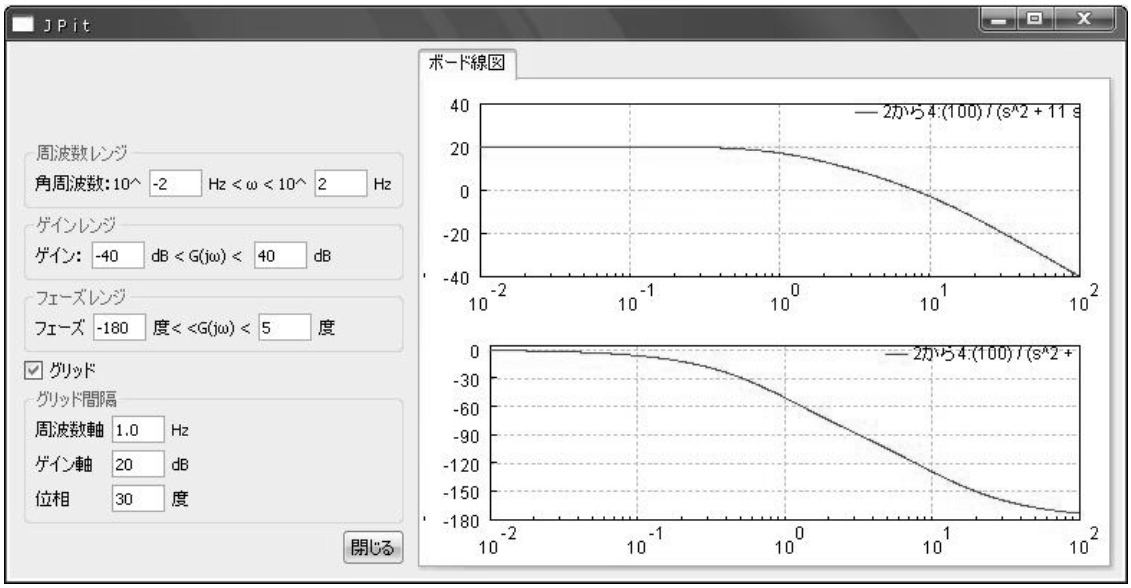


図 5.3 $K_P = 10$ のときの開ループゲイン線図を Jpit を用いてプロットした図

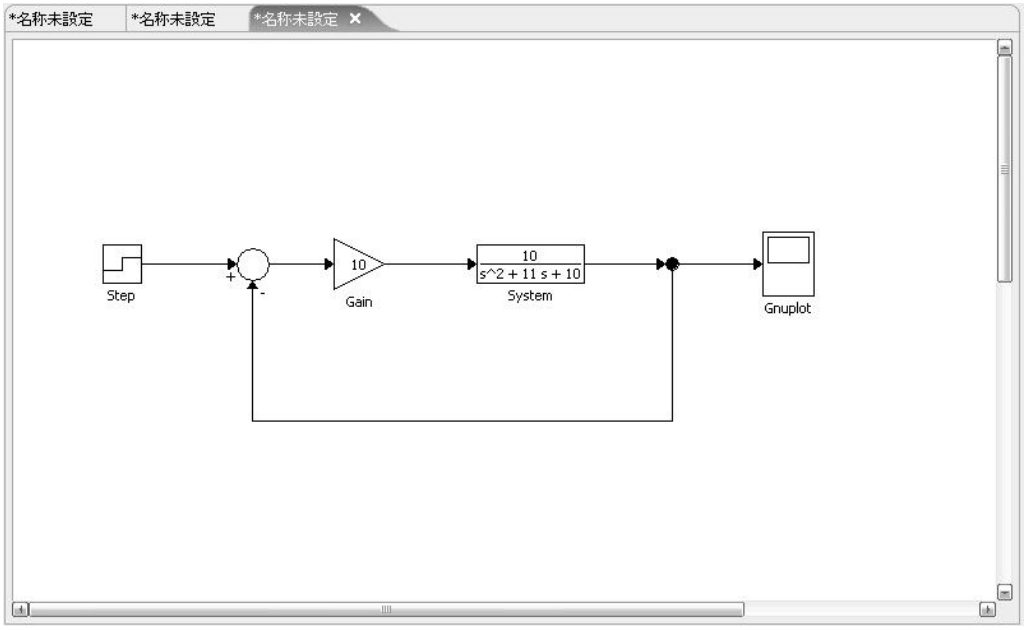


図 5.4 $K_P = 10$ のときの閉ループ系

図 5.7 より，低周波域において PI 補償の方が開ループゲインが高く，特に I 補償の効果で $|L_{PI}(0)| = \infty$ であり 1 型の制御系となっている。したがって定常偏差は 0 になるはずである。本ツールでステップ応答を求めたものが**図 5.8** であるが，たしかに定常偏差が 0 になっていることがわかる。

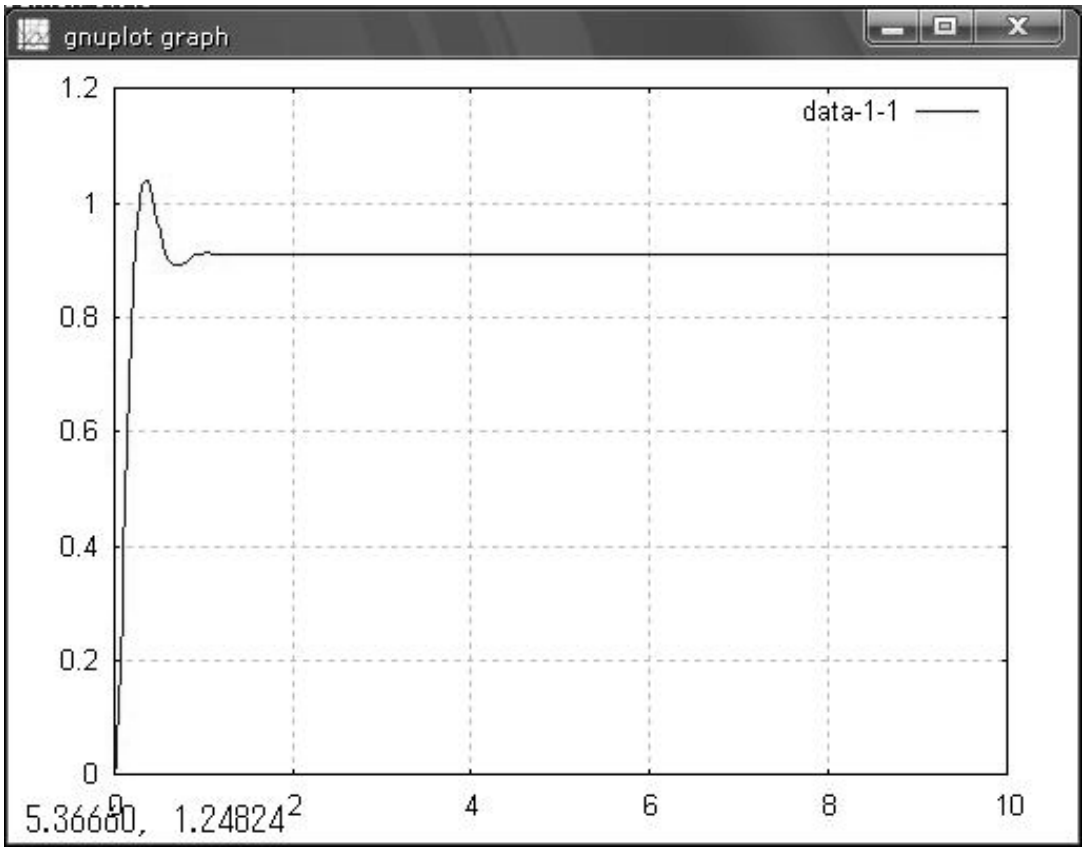


図 5.5 $K_P = 10$ のときのステップ応答を gnuplot を用いてプロットした図

5.2 ロバスト制御問題

5.2.1 混合感度問題 [1, pp105-106]

図 5.9 の混合感度問題の枠組で H_∞ サーボ問題を考える。このとき

$$G_o = \begin{bmatrix} 0 & W_T P \\ I & P \end{bmatrix}$$

である。

$$P(s) = \frac{1}{s-1},\ W_T(s) = \frac{s+1}{10},\ W_s(s) = \frac{5}{s}$$

としたときに Jamox を使ってモデル化したときのモデルは**図 5.10** である。得られた系の感度関数と相補感度関数のゲイン線図を**図 5.11** に示す。

このように，本ツールを用いてモデル化 · 解析を行った結果は例題で示した結果と同じであり，ロバスト制御問題の設計においてもツールは有効であるといえる。

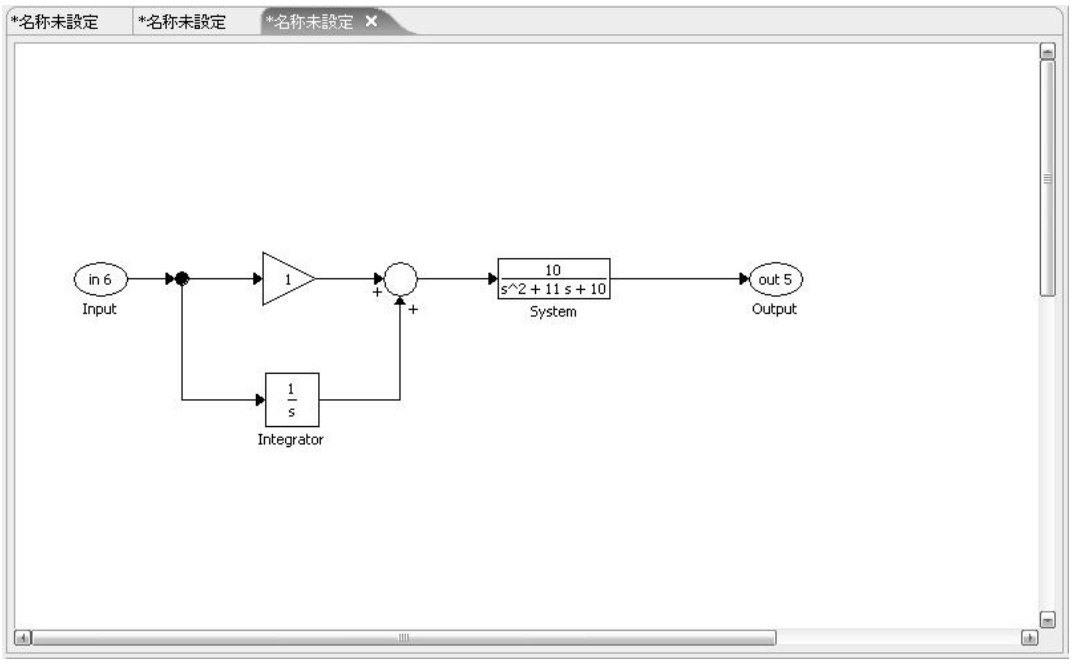


図 5.6 $K_{PI}(s) = \frac{s+1}{s}$ のときの開ループゲインを求めるためのブロック線図

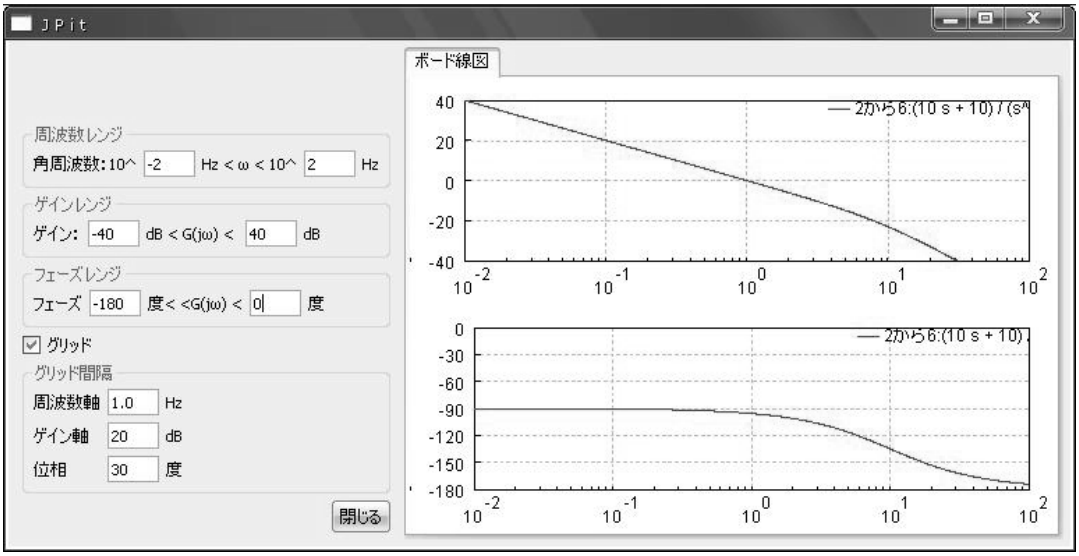


図 5.7 $K_{PI}(s) = \frac{s+1}{s}$ のときの開ループゲイン線図を Jpit を用いてプロットした図

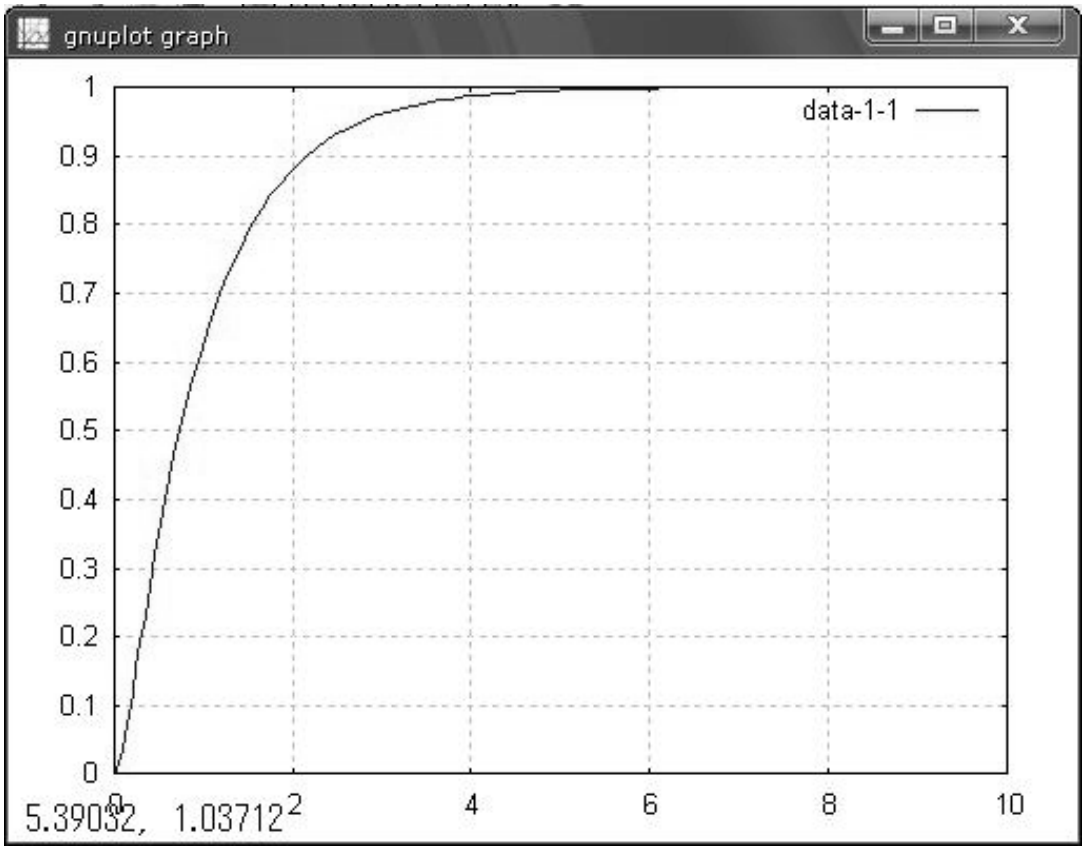


図 5.8 $K_{PI}(s) = \frac{s+1}{s}$ のときのステップ応答を gnuplot を用いてプロットした図

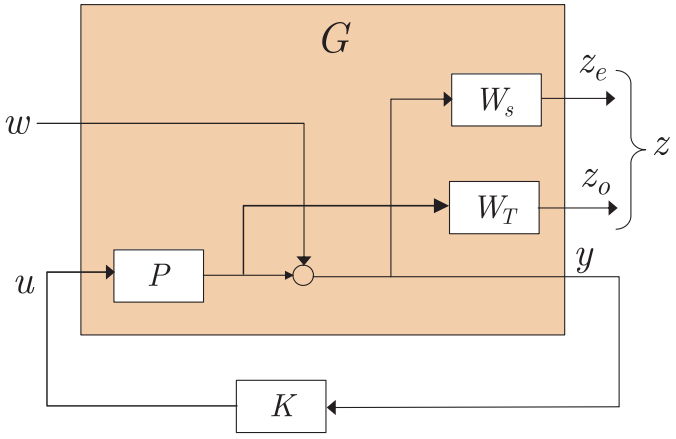


図 5.9 混合感度問題

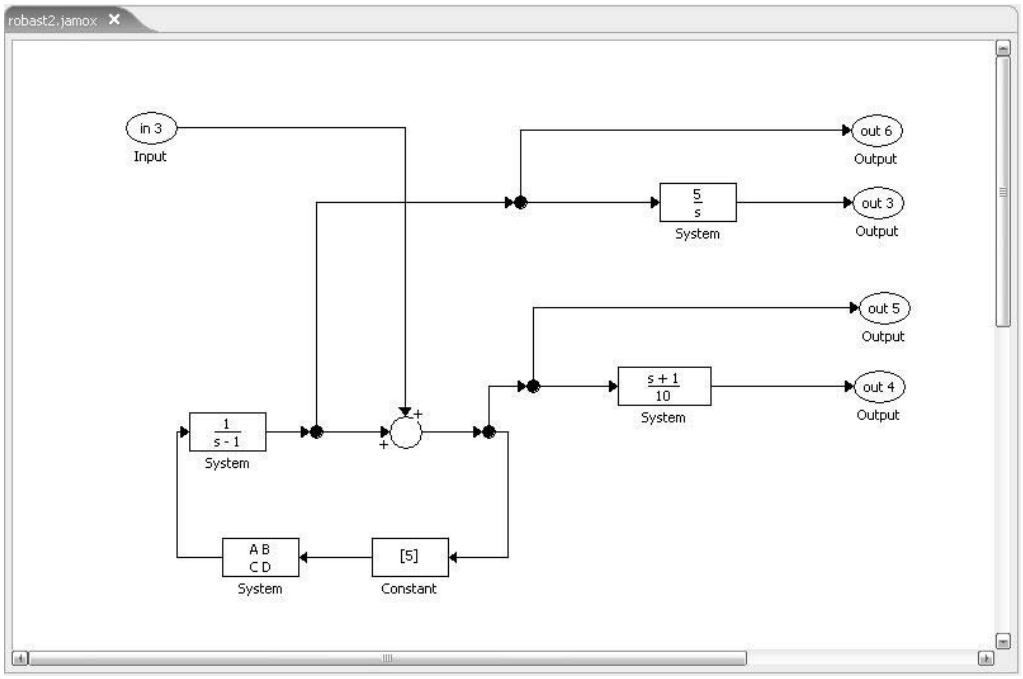


図 5.10 図 5.9 を Jamox でモデリングした画面

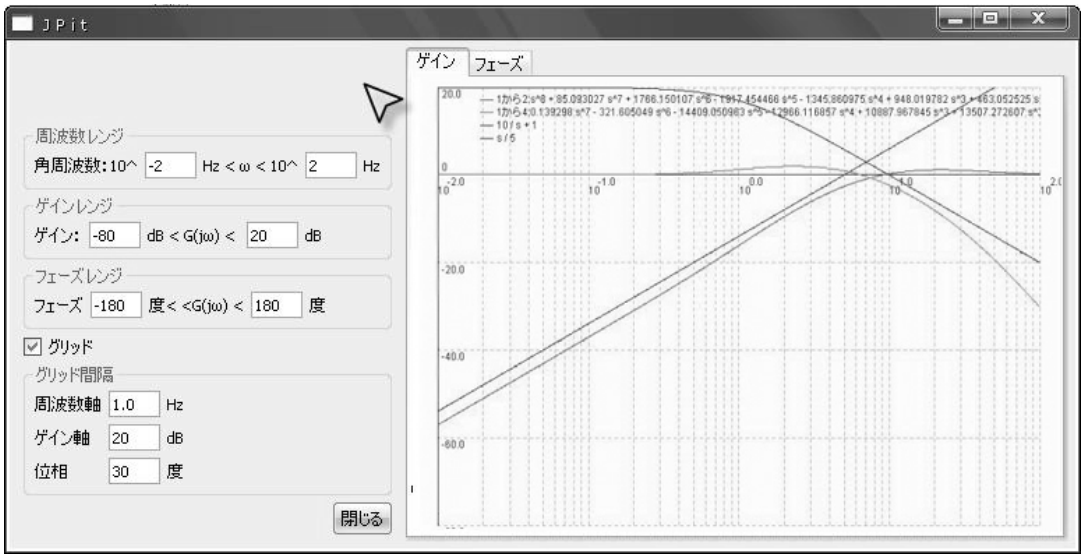


図 5.11 感度関数と相補感度関数のゲイン線図

第 6 章

結論

本研究では，ブロック線図を用いた制御系のモデリング · シミュレーションプラットフォームを開発した。

古典制御問題で PI 補償問題の設計に Jamox を適用し，その有効性を確認した。また，混合感度問題に対してもモデル化 · 解析を行いその有効性を確認することでロバスト制御問題においても Jamox が有効であることを確認した。

ブロック線図のデータを XML 形式で保存することで，データを再利用可能にした。また，データバインディング技術を利用することで，構造の変化に容易に対応でき，同時に厳格な定義，それに伴う妥当性の検証もできることを示した。

シミュレーション結果 (グラフ) の表示では Jpit や gnuplot を利用することで，他ツールとの連携ができるということを示した。他にもユーザーが使用したいアプリケーションと使い方を登録することで，好きなアプリケーションに結果を渡すことができるように機能を拡張できる。

謝辞

本研究を進めるに当たって，さまざまな方にお世話になりました。その中でも研究の全ての面において特に綿密なご指導を頂いた，古賀雅伸助教授に深く感謝致します。

研究だけでなく様々な面で助言やアドバイスしてくれた矢野健太郎氏に深く感謝致します。Jamox について，また，JAXB を利用するにあたり，様々なことをサポートしてくれた竹井佑介氏に感謝致します。何も知らない研究室配属当初のパソコンのセットアップ等をお世話してくれた岸田和也氏に感謝致します。先輩として，色々分からないことなどお世話してくれた木村裕樹氏，松永隆徳氏，森宗翔吾氏に感謝致します。Jamox のユーザとして，色々なことを報告してくれた古賀崇史君に感謝します。隣接行列やシミュレーションのアルゴリズムについて教えてくれた中村圭之介君に感謝します。MATX エンジンを利用するにあたり質問や要望に応えてくれた田中俊行君に感謝します。そして，いつも楽しい研究室にしてくれた，赤峰義明君，有馬達郎君に感謝します。

最後に，経済的に支えてくれた家族に深く感謝致します。

付 録 A

SWT と JFace

Eclipse は高度な統合ツールの開発のために必要な様々な機能を持つ安定したプラットフォームを提供するために寄付された，オープンソースの普遍的なツールプラットフォームである。Eclipse プロジェクトは IBM、Object Tchnology International(OTI) や幾つかの企業によって 2001 年に開始された。そして，Eclipse は 100ヶ国以上の多数の開発者を魅了し，300 万件以上のダウンロードが行われた。

Eclipse プラットフォームは，ほとんどのツールビルダーから必要とされるフレームワークや一般的なサービスの集合を定義している。その中で最も重要な物の一つは，移植可能なネイティブ · ウィジェット · ユーザーインターフェースである。Standard Widget Toolkit(SWT) は移植可能なネイティブ · ユーザーインターフェースのサポートと同様に，ウィジェットとグラフィックスのための OS 独立な API を提供する。

JFace は SWT に基づいたピュアな Java UI フレームワークであり，多くの一般的な UI プログラミングを扱う事ができる。続く節で SWT と JFace の詳細について述べる。

A.1　Standard Widget Toolkit

SWT は Java の AWT と Swing に類似しているが，豊富なネイティブ · ウィジェットを使用している点が異なっている。AWT のウィジェットはネイティブ · ウィジェットを直接用いて実装されているので，移植するためには全ての種類のウインドウシステムの共通点を取る必要がある。例えば Windows はツリーウィジェットをサポートしているが，Motif はそうではない。その結果，AWT はツリーウィジェットを持つ事ができない。Swing はほとんどの種類のウィジェットをエミュレートする事で，この問題に対処している。しかしながら，このエミュレーションによる解決はいくつかの深刻な欠点を持っている。まず，エミュレートされたウィジェットはルック · アンド · フィールがネイティブ · ウィジェットのものよりも劣っており，ユーザとエミュレートされたウィジェットとの対話はネイティブ · ウィジェットとの対話とはかなり異なる。次に Swing は改善されてはいるが，Swing のユーザーインターフェースはまだ不十分である。

SWT は，若干異なる方法を採用している。SWT はサポートされているウインドウシステムで利用可能な共通の API を定義している。それぞれのネイティブウインドウシステムのために，SWT の実装はネイティブ · ウィジェットが利用できる場所全てでネイティブ · ウィジェットを利用している。もし利用できるネイティブ · ウィジェットが無い場合には，SWT の実装はエミュレーションを行う。前に述べたように Windows はツリーウィジェットを持っ

ているので，SWT は Windows のネイティブのツリーウィジェットを使っている。Motif はツリーウィジェットを持っていないので，SWT の実装は適切なエミュレートされたツリーウィジェットを提供する。この方法では，SWT は一貫性のあるプログラミングモデルを全てのプラットフォームで保持し，どんな基本的なネイティブ · ウィンドウ · システムも利用する。

SWT は基本的なネイティブ · ウィンドウ · システムとしっかりと統合されている。SWT はプラガブルなルック · アンド · フィールをサポートしていないが，多数の計り知れない機能を提供している。例えば，(ドラッグアンドドロップの様な) ネイティブ UI の相互作用や，(Microsoft Word, Acrobat Reader 等の Windows の ActiveX コントロールの様な)OS 固有のコンポーネントへのアクセスがある。

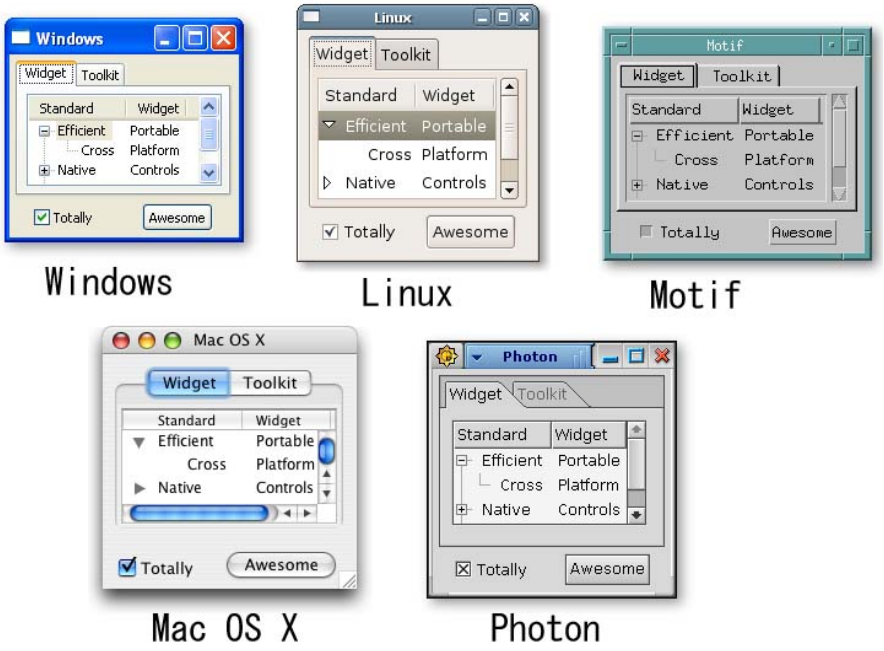


図 A.1 各 OS のルック · アンド · フィールの違い

SWT は開発者に Java を用いてネイティブ · ユーザーインターフェースを作成する事を可能にする。しかしながら，Windos や Linux や他のプラットフォームでユーザーインターフェースを開発してきた経験を持つほとんどのプログラマは，GUI を開発することは複雑で時間のかかるプロセスであることを知っている。Java でネイティブ · ユーザーインターフェースを作成することも，例外ではない。幸運にも，Eclipse はネイティブ · ユーザーインターフェースのプログラミングプロセスを容易にする，JFace と名付けられた UI ツールキットを提供している。

A.2 JFace

JFace は SWT を使用して多数の一般的なユーザーインターフェースプログラミングの問題を扱うために実装された UI ツールキットである。JFace はその API や実装に依存しないウインドウ・システムである。JFace は，SWT を隠さずに協調して動作するように設計されている (**図 A.2** 参照)。

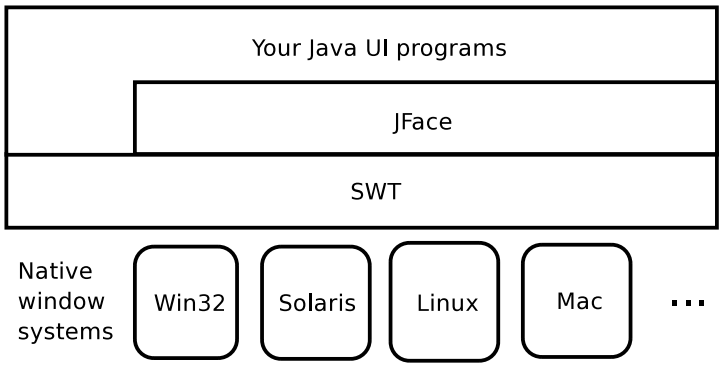


図 A.2 JFace

JFace は次の構成要素を提供している。

- イメージレジストリとフォントレジストリ
  イメージレジストリとフォントレジストリは開発者が OS の資源を管理するのを助ける。
- ダイアログとウィザード
- 長い時間のかかる操作を報告するプログレス
- アクションの構造
  アクションの構造は，ユーザーのコマンドをユーザーインターフェースの正確な場所から分離する。アクションはユーザーがボタンやメニューアイテムやツールバーアイテムから実行する，ユーザーのコマンドを表現する。それぞれのアクションはラベルやアイコンやツールチップ等のそれ自身の必要不可欠な UI ウィジェットを定義する。そして，それはアクションを提示するために適切なウィジェットが使われる。
- ビューワーとエディター
  ビューワーとエディターはいくつかの SWT ウィジェットのためのモデルベースのアダプタである。一般的な振る舞いと高レベルのセマンティクスはそれらの SWT ウィジェットのために提供される。

付 録B

JAXB[2][3]

Jamoxは，XML形式のファイルを扱うため，Java SE 6 MustangにJAXB 2.0を用いている。Java言語でXMLを扱うAPIはDOM[19]やSAX等が存在するが，データとして厳格に定義されたXMLドキュメントを扱う際，プログラムは煩雑になり修正が容易ではない。JAXBはXML Schemaドキュメントに記述されたデータ型に対応したJavaクラスファイルを用いるため，XMLドキュメントに対する不正なデータの追加·編集することは不可能であり，データとしてのXMLドキュメントを安全に使用することが可能である。

JAXB(Java Architecture for XML Binding)は，Java/XML間のデータ·バインディングのためのAPIである。このAPIのヴァージョン1.xは，JWSDP(Java Web Services Developer Pack)の一部として提供されている。本論文では，明記しない限りヴァージョン2.0を指す。

Java/XML間のデータ·バインディングとは，Javaのデータ(クラスやインターフェースの定義，及びそのインスタンス)とXMLのデータ(スキーマ定義，及びXMLドキュメント)の相互変換を行うことを指す。JAXBの特徴を以下に示す。

- XML Schemaをフルサポート
- Javaから，XML Schemaベースのスキーマ言語への自動生成·マッピングが可能(この逆は，JAXB 1.xで既にサポートされていた)
- セマンティクスの違いを吸収するためにアノテーションを利用している

B.1　JAXBの概念

図B.1にJAXBの概念を示す。

JAXBは，XMLドキュメントをJavaのクラスやオブジェクトにバインドする(対応づける)という概念の上に成り立っている。つまり，XML Schema等のスキーマ言語で記述したスキーマドキュメントによってXMLドキュメントのデータ構造が決まり，そのデータ構造やデータの制約条件をスキーマから参照し，そのスキーマに従ったXMLドキュメントを扱うためのクラスを生成する。結果として，そのスキーマドキュメントに従って作成されたXMLドキュメントは，Javaのオブジェクトとしてアプリケーションから扱うことが可能となる。

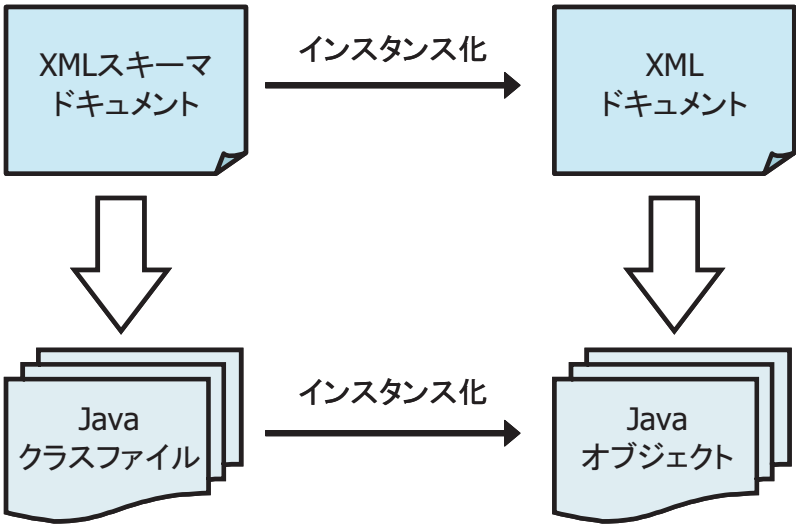


図 B.1 JAXB の概念

B.2 他の API との違い

DOM や SAX 等の他の XML を扱う API を使用して開発を行う場合，その自由度は非常に高い。しかし，その半面 XML ドキュメントの構造や要素 · 属性等に依存したコードを逐一記述する必要がある。また，XML データの構造に依存する繰り返し処理に関しては自分で汎用的に開発する必要があり，非常に煩わしいコーディング作業が発生する。これに対し，JAXB を利用する場合は，あらかじめスキーマに沿った形で java クラスが生成されることになり，要素 · 属性に関しての情報を改めて記述する必要がない。そのため，生成された Java クラスを扱うアプリケーション側にエラーがない限り，スキーマドキュメントに基づいた厳格な XML ドキュメントを扱うことが出来る。

B.3 JAXB を用いたプログラミング

JAXB を利用する場合，一般的な開発の流れは**図 B.2** のようになる。

まず最初に，XML Schema ドキュメントを JAXB が提供するコンパイラに読み込ませる。これにより，XML Schema ドキュメントにより定義されているデータ型に従った Java クラスが生成される。

生成された Java クラスを利用することで，通常の Java プログラミングを行うのと同様の操作で XML ドキュメントを扱うことができる。また，XML Schema か Java クラスのどちらかに変更が生じた場合でも，すぐにお互いを厳格に対応づけることが可能であり，効率的なコーディングが可能になる。

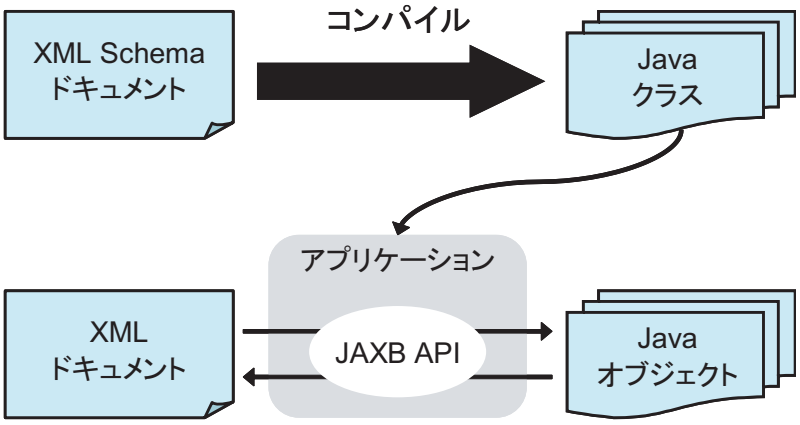


図 B.2 JAXB を用いたプログラミングの流れ

参考文献

[1] SICE セミナー「ロバスト制御入門」 in 北九州, 11 2003. カタログ番号 03 SE 0006.

[2] Sun Microsystems. *JAXB*. http://java.sun.com/webservices/jaxb/.

[3] 竹井佑介. 制御系のモデリング言語 csml を用いた制御系設計に関する研究. Feb. 2008. 2007 年度修士論文.

[4] ロバート・C・マーチン. アジャイルソフトウェア開発の奥義. ソフトバンク パブリッシング, 2004.

[5] 古賀雅伸. Linux·Windows でできる MATX による数値計算. 東京電機大学出版, 2000.

[6] *The MathWorks - Matlab*
. http://www.mathworks.com/products/matlab/.

[7] 松木毅. Java 数値計算パッケージの開発と制御系設計への応用. Feb. 2003. 2002 年度卒業論文.

[8] 森智宏. グラフで設計仕様を与えることができる制御系設計支援ツール. FIT2004 第 3 回情報科学技術フォーラム, pp. 641–642, 2004.

[9] *The MathWorks - Simulink*
. http://www.mathworks.com/products/simulink/.

[10] *Dynasim*
. http://www.dynasim.se/.

[11] WinCSCAD サポートページ
. http://www.is.osaka-kyoiku.ac.jp/~takeuchi/wincscad/.

[12] *Scicos Homepage*
. http://www.scicos.org/.

[13] *BrainBox*
. http://brainbox.sourceforge.net/en/index.php.

[14] *eclipse RCP*. http://www.eclipse.org/home/categories/rcp.php.

[15] 山本祐, 原辰次. サンプル値制御理論—システムとその表現 1. Vol. 6, No. 10, pp. 1–8, 1999.

[16] 中村圭之介. 隣接行列を用いた制御系のモデリング · シミュレーションに関する研究. Feb. 2008. 2007年度卒業論文.

[17] 田中俊行. 数値計算エンジン · インタプリタの開発. Feb. 2008. 2007年度卒業論文.

[18] 藤田 政之 共著杉江 俊治. フィードバック制御入門. コロナ社, 1999.

[19] W3C. *DOM*. http://www.w3.org/DOM/.

[20] 奥井康弘 共著中山幹敏. 改訂版 標準ＸＭＬ完全解説 (上). 株式会社技術評論社, 2001.

[21] Brett McLaughln. Java & XML 第2版. 株式会社オライリー・ジャパン, 2002.

[22] 美田 勉 共著中野 道雄. 制御基礎理論 [古典から現代まで]. 昭晃堂, 1999.

[23] 大村忠史. Java GUI プログラミング SWT 編. 株式会社カットシステム, 2003.

[24] 原辰次, 小林史典. 制御系 cad のための結合計算アルゴリズム. システムと制御, Vol. 29, No. 2, pp. 105–114, 1985.

[25] 古賀雅伸, 筒井勇介. ブロック線図を用いた制御系のモデル化 · シミュレーションプラットフォームの開発. Feb. 2007. 2006年度修士論文.

[26] 内藤毅. Mathml による数式表現に基づく制御系モデリング言語 csml. Feb. 2004. 2003年度修士論文.

[27] *World Wide Web Consortium*
. http://www.w3c.org/.

[28] *XML Consortium*
. http://www.xmlconsortium.org/.

[29] *SVG(Scarable Vector Graphics)*
. http://ginger.cs.inf.shizuoka.ac.jp/~jun/GS/SVG/.

[30] *Modelica Portal*
. http://www.modelica.org/.

[31] *The MathWorks: Online Documentation (Help Desk) - Release 13 (Japanese)*
. http://www.mathworks.com/access/helpdesk/jhelp/helpdesk.shtml.

[32] *Introduction to SVG*
. http://wwws.sun.com/software/xml/developers/svg/.

[33] *Gnuplot Central*
. http://www.gnuplot.info/.

[34] *Eclipse.org Main Page*. http://www.eclipse.org/.

[35] 月刊ジャバワールド, 第 77 巻. 株式会社アイ · ディ · ジー · ジャパン, 10 2003.

[36] 「The Java3D API 仕様」.

[37] 名取亮, 長谷川英彦, 櫻井鉄也, 桧山澄子, 周偉東, 花田孝郎, 北川高嗣. 数値計算法. pp. 113–115, 2000.

[38] 青木健太. 制御系シミュレーションのためのオブジェクト指向結合計算パッケージの開発. 2005. 2004 年度卒業論文.