MATX 使用の手引き (1999 年 9 月 20 日)

(東京工業大学 制御システム工学科 古田研版)

東京工業大学　工学部　制御システム工学科　西村 修
東京工業大学　情報理工学研究科　情報環境学専攻　古賀 雅伸

1　はじめに

本文は，初めて MATX を使う人に対して，その最初の手引きとなることを期待して書かれたものです。具体的には，東京工業大学 工学部 制御システム工学科 古田研究室新人教育の際に制御系のシュミレーションで MATX を用いる時のイントロの為のものです。ただし，MATX にはきちんとしたマニュアルが存在しますので，マニュアルを参照し，実際に使いながら覚えていく方が良いでしょう。なお，この文書はある程度の C 言語の知識，UNIX の知識を前提とします。

2　MATX について

MATX は東京工業大学情報理工学研究科情報環境学専攻助手の古賀さんが作られたものです。C 言語で書かれており，古田研周辺のほとんどのマシンに移植されています。MATX は制御工学を学ぶ人には大変便利に出来ているプログラム言語です。制御系設計に欠かせない行列計算はもちろん，シミュレーションの為の関数も多数用意されています。MATX は 1989 年以降，現在に至るまで古田研の制御系設計のためのプログラム言語として主に使われてきました。数年前から，MATX を使ってシミュレーションを行ない，それをもとに実験を行なうというスタイルが主流を占め，理論を行なう人もシミュレーションに用いるようになってきています。

なお，MATX のソースファイルのデフォルト拡張子は.mm です。MATX のソースファイルのことを **MM-** ファイルと呼びます。ちなみに，制御系解析で有名なソフト Matlab のソースファイルは **M-** ファイルと呼びます。

3　起動方法

MATX にはインタプリタとコンパイラがあり，制御系設計などのそれほどマシンパワーを必要としないものにはインタプリタを，シミュレーションなどで大変な時間を要する時にはコンパイラを用いることが出来，xplot[1], gnuplot などと組み合わすことによって快適な作業空間を構築することが出来ます。

[1] X ウィンドウで使用できる GUI プロットツール

3.1　インタプリタ matx

インタプリタは

```
% matx
```

と起動します。そうするとオープニングメッセージのあとで，

```
MaTX(1)
```

が表示されます。これが MATX のコマンドラインとなります。コマンドラインでも Emacs キーバインドで自由にエディット出来ます。

また，終了するには，MATX コマンドラインで

```
MaTX(2) exit
```

とします。

3.2　コンパイラ matc

MATX でコンパイルを行うには matc を用います。MATX のソースは.mm という拡張子を持っていますが，matc は，これから実行可能なプログラム (UNIX の場合は拡張子はなく，DOS/Windows のの場合は拡張子は.exe) を生成します。例えば simu.mm というファイルを作っていたとすれば，

```
% matc simu.mm
```

で simu(DOS/Windows の場合 simu.exe) という実行可能プログラムが出来上がります。

3.2.1　C 言語のソースの生成

matc をオプション -mm 付きで実行すると MATX のソースを C 言語のソースに翻訳して，.c というファイルに変換します。例えば simu.mm というファイルを作っていたとすれば，

```
% matc -mm simu.mm
```

で simu.c という C 言語のソースが出来上がります。具体的には MATX で用いるライブラリなどのヘッダファイルがインクルードされ，行列関数などが，C 言語に準拠したものに翻訳されます。

また，matc をオプション -c 付きで実行すると MATX のソースと C 言語のソースからオブジェクトファイル.o に変換します。例えば simu.mm というファイルを作っていたとすれば，

```
% matc -c simu.mm
```

で simu.o(DOS/Windows の場合 simu.obj) というオブジェクトファイルが出来上がります。
C 言語のソースやオブジェクトファイルから実行可能プログラムを生成することもできます。例えば simu.c という C 言語のソースを作っていたとすれば，

```
% matc simu.c
```

とすれば simu という実行ファイル (DOS/Windows の場合，simu.exe) が出来上がります。概算でインタプリタよりもコンパイラの方が 4，5 倍スピードが速くなっています。したがって，シミュレーションのバグをインタプリタでとり，その後に実際にシミュレーションする時にはコンパイラを用います。

3.2.2 Makefile の利用

C 言語の演習で Makefile の書き方を知ったと思います。Makefile を用いる時には.o というファイルを gcc -c コマンドで作りました。 -c が .o を作るスイッチだったわけです。 MATX で分割コンパイルを行う場合，.o というファイルを作るコマンドが matc -c です。具体的には，

```
% matc -c simu.mm
```

で simu.o というファイルが出来ます。あとは最後に

```
% matc simu.o otherfiles.o
```

とすべての.o ファイルを matc でリンクすればいいです。実行ファイルの名前は最初に書かれた.o ファイルの名前になります。

3.3 行列エディタ mated

色々な行列を作る為のエディタが用意されています。これはコマンドラインで

```
% mated A.mat
```

などとすると，A.mat という名前のファイルに行列をエディットすることが出来ます。行列エディタは Emacs キーバインドで書くことが出来ます。行列エディタの終了は q です。

4 MATX プログラムの書き方

今まで，コマンドを数々並べてきました。では実際にどういうプログラムを書けばよいのかをこれから説明します。

4.1 C 言語との類似性

MATX は基本的には C 言語とほとんど同じです。 C 言語と少し違うところは，変数の型，配列，ファイル入出力などです。ただし MATX にポインタは存在しません。関数に複数の返り値を持たせたい場合は返り値に List を使うと良いでしょう。その詳細はマニュアルを参照して下さい。

4.1.1 変数の型

変数の型は以下の 10 種類です。 pascal に近いといえるでしょう。 int が Integer，double, float が Real という感じです。この文書では文字列，指数，リストについては取り扱いません。

Integer	整数
Real	実数
Complex	複素数
String	文字列
Polynomial	多項式
Rational	有理関数
Matrix	行列
Array	配列
Index	指数
List	リスト

変数のうち，関数引数として取る場合に，行列，多項式，有理関数，リストが参照渡しされます。つまりポインタで渡されるのと同じです。そのほかの変数は値渡しされます。この区別は重要です。
また，注意することは，実数はすべて小数点をつけることです。また変数の型は C 言語と同様，型変換することができます。行列どうしの計算で結果が 1×1 となってもそれはあくまで行列でありスカラにするには型変換が必要です。

4.1.2 制御構造

if，for，while，do，switch などは全く同様に使えます。ただし，{ } の省略は出来ません。

4.1.3 配列

MATX の配列は行列とは区別されていますが，とりあえずは配列を使う場面ではすべて行列を使って構いません。ただし，C 言語と表現が違っています。行列の使い方は次節に譲ります。

4.1.4 関数定義

関数定義の方法は C 言語と少し違います。関数の定義は

```
Func void main()
{

}
```

というように最初に Func を使います。関数の中で用いるほかの関数は，C 言語同様に普通に書きます。これは後のシミュレーションの例で分かるでしょう。

4.1.5 行列

MATX で一番用いることの多い型がこれです。MATX はほとんどの行列の計算をサポートします。また，普通の数式で表現出来る為，ソースが非常に見やすくなります。
行列は次のように書きます。

```
A = [[ 1 2 3 ]
     [ 4 5 6 ]
     [ 7 8 9 ]];
```

C 言語と同様に空白や改行は意味をもたないので，詰めて書いても構いません。四角ブラケット “[”, “]” で行の始めと終りを示し，更に外側で行列の始めと終りを示します。成分は “(”, “)” で参照します。

```
A(1,1) = 1; A(1,2) = 2; A(1,3) = 3;
A(2,1) = 4; A(2,2) = 5; A(2,3) = 6;
A(3,1) = 7; A(3,2) = 8; A(3,3) = 9;
```

でも同じことです。C 言語では “[”, “]” で配列の中身を参照しますが，MATX では “(”, “)” ですので，注意して下さい。またその番号も C 言語では 0 から始まりますが，MATX では 1 から始まります。これらの行列は行列エディタ mated でも書けますし，matx のコマンドラインで

```
MaTX(2) read A
```

とすれば書けます。Matrix 宣言されている変数はスカラと同様に式に書けます。例えば状態方程式は次の 1 行で書けてしまいます。

```
A * x + B * u
```

これらの例はシミュレーションの節で述べます。

5 多項式関数

MATX は制御系解析で用いられる有理関数の取り扱いを簡単にするために，多項式関数が用意されています。MATX の中では，多項式の変数は Polynomial() で定義します。伝達関数を表すのには次のようにします。

```
Polynomial s;
Rational G;
Matrix A, B, C, D;

s = Polynomial("s");
G = 1/(s^2 + 2 * s + 1);
{A,B,C,D} = tfm2ss([G]);
```

この例は伝達関数を状態空間表現する例です。

5.0.6 ファイル入出力

C 言語ではファイルの入出力などはかなり大変でした。printf 文などもその表示形式を指定する必要があり非常に面倒でした。しかし，MATX はそのほとんどを print と read で済ますことが出来ます。行列 A などをファイルに落したい時には，

```
print A >> "A.mat"
```

とするだけで A.mat というファイルにそのデータを出力できます。また，

```
read A << "A.mat"
```

とするだけで，A.mat というファイルの中身を行列 A に代入することが出来ます。

5.1 行列計算の基本

行列は普通に計算することが出来ます。MATX は強力な字句解析，構文解析機能を有し，ほとんど必要な行列計算をスカラと同じ感覚で行なうことが出来ます。したがって，行列の成分の計算のための for ループを一切用いる必要がないように作られています。

```
Matrix a, b, x, dx, u;

a = [[ 1 2 3 ]
     [ 4 5 6 ]
     [ 7 8 9 ]];

b = [ 1 2 3 ]';

x = [1 1 1]';
u = [1];
dx = a * x + b * u;
```

この例のように，Real や Integer などの変数と同じ様に普通に計算することが出来ます。例えば基本的な演算は以下の通りです。Matrix A としたとき，

転置	A'
逆行列	A~
単位行列	I(3,3)
零行列	Z(3,3)
対角行列	diag(1.0, 2.0, 3.0)
ランク	rank(A)
固有値	eigval(A)

6 MM- ファイルの実行方法

ユーザーが作った MM- ファイルをインタプリタ matx で実行する方法は次の 2 種類です。

1. MATX のコマンドラインで

   ```
   MaTX(1) load "file.mm"
   ```

   とロードコマンドで読み込み，それを実行するには

   ```
   MaTX(2) main()
   ```

   と関数名を打てば実行できます。void 型の関数で引数が無いときは

   ```
   MaTX(2) main
   ```

   のように括弧を省略できます。

2. 最初から読み込む。UNIX のコマンドラインで，matx を実行するときに，

   ```
   % matx file.mm -
   ```

   とします。- を付けると，matx が file.mm を読み込んだ後，MATX コマンドラインになってくれるので，そこで main() と実行すれば良いでしょう。これが面倒くさければ，ファイルの最後に実行してもらいたい関数名を書きます。例えば後に示すシミュレーションでは最後に main(); としておけば実行してくれます。

コンパイラの使い方は先に述べたように，C 言語と全く同様です。

7 標準提供関数

MATX が標準で提供する関数には。MATX 内部に組み込まれている組み込み関数と機能拡張のためユーザが定義するユーザ定義関数があります。

7.1 組み込み関数

組み込み関数とは MATX に最初から組み込まれている関数です。MATX のコマンドラインで

```
MaTX(1) what
```

と打つと最初に出てくるのがその関数です。それぞれに関数の働きは，

```
MaTX(1) help 関数名
```

で見ることが出来ます。

7.2 ユーザ定義関数

ユーザ定義関数とは MATX の `MaTX/inputs` の中の MM-ファイルのことです。これらは制御系解析に必要な役に立つ関数群が入っています。例えば，可制御行列 V を求める関数 `CoAbleMat()` などを代表とし，このほか当然のごとくリカッチ方程式を解く関数 `Ric()` や，オブザーバを実現する関数 `Gopinath()` などもありますし，後に説明する H_∞ 問題を解いてくれる関数 `Hinf()` などもあります。これらの説明も組み込み関数と同様に `help` で見ることが出来ます。これらの MM- ファイルはテキストファイルですので，このディレクトリにいけば，その中身を見ることが出来ます。

実際にユーザが色々な関数を作っていく上で，このディレクトリに収まっている関数を見本にするといいでしょう。

8 シミュレーションの基礎

この節ではシミュレーションの基礎の仕組みについて説明します。最近では，古田研でシミュレーションをする場合は MATX を用いることが多くなってきました。それは MATX に微分方程式を積分計算してくれる数々の関数が用意されているからです。

では，まずシミュレーションの雛型から説明します。

8.1 diff_eqs(), link_eqs()

MATX には数々の積分計算関数が用意されていますが，基本的には次のプログラムのように使います。プログラムの `main()` 関数の 9 行目の `Ode` というのがその形です。

```
Func void main()
{
  Real dt;
  Matrix x0;
  Array TC, XC, UC;
  Matrix diff_eqs(), link_eqs();

  dt = 0.01;

  x0 = [1 1 1]'; // 初期状態
  {TC, XC, UC} = Ode(0.0, 5.0, x0,
                     diff_eqs, link_eqs);

  print [[TC][XC][UC]] >> "cont.mat"
}

Func Matrix diff_eqs(t, x, u)
  Real t;
  Matrix x, u;
{
  Matrix dx, A, b;

  A = [[ 0,  1,  0]
       [ 0,  0,  1]
       [-2, -3, -4]];

  b = [0 0 1]';

  dx = A * x + b * u;
  return dx;
}

Func Matrix link_eqs(t, x)
  Real t;
  Matrix x;
{
  Matrix u, f;

  f = [ -1, -1, -1];
  u = f * x;
  return u;
}
```

関連する変数としては，状態 `x`，入力 `u`，時間 `t`，状態の微分 `dx` とが必要です。これらは関連関数の `link_eqs()`, `diff_eqs()` に必要です。

積分関数 (`Ode()` など) は少なくとも 5 個引数をとります。最初から開始時間，終了時間，初期状態,`diff_eqs`，`link_eqs` となります。

この `diff_eqs()` はシステムの状態の微分値を返す関数です。言い変えれば，時間 `t` と状態 `x` と入力 `u` から，状態の微分 `dx` を返す関数のことです。それぞれの変数は `diff_eqs()` の引数の欄を書き換えると，変えることが出来ます。例えば `Ode(0.0, 5.0, x0, diff, link)` と書くならば，`diff`, `link` というのがそれぞれの関数となります。

`diff_eqs()` の型が `Matrix` であることに注意して下さい。`diff_eqs()` の引数の 3 つが重要です。それぞれの変数は `Ode()` が渡していると思って下さい。

`link_eqs()` はシステムへの入力を返す関数です。言い変えれば，時間 `t` と状態 `x` からシステムへの入力 `u` を決め

る関数のことです。link_eqs() の型も Matrix です。この link_eqs() の 2 つの引数も Ode() が渡していると思って構いません。

Ode() の返す値は例をみると {TC, XC, UC} となっています。これは Array で宣言されていますが，それぞれ，時間，状態，入力の時系列データとなっています。したがって，最後の文の

```
print [[TC][XC][UC]] >> "cont.mat"
```

では 3 個の状態と 1 個の入力を持つシステムなので，1 行目に時間が，2 行目から 5 行目までに状態が，6 行目に入力があるデータが行列として cont.mat というファイルに書き出されます。ファイルの出力形式は転置してから出力されますので，ファイルの中では列毎に各成分が並びます。

8.2 動的補償器

これで大体の基本は分かったと思います。この例では状態フィードバックなので static なコントローラでした。これが動的なコントローラだったり，オブザーバが入っていたり，Adaptive のパラメータ更新則も入っていたらどうなるでしょうか？

答えは，すべて diff_eqs() です。前の例では diff_eqs() にはプラントの式しかありませんが，あくまで MATX の積分関数は diff_eqs() の返す微分値と link_eqs() の返す入力から積分しているに過ぎないので，微分方程式で表されるものはすべて積分してしまえば良いことになります。

例えば，先のシステムにおいて，制御の為に必要なパラメータが次の微分方程式で表されているとします。

$$\dot{p} = -G \times (x - a)$$

この式は微分方程式ですから diff_eqs() に書いてしまえばよくて，

```
Func Matrix diff_eqs(t, x, u)
  Real t;
  Matrix x, u;
{
  Matrix A, b;
  Matrix dx, dp;

  A = [[ 0,  1,  0]
       [ 0,  0,  1]
       [-2, -3, -4]];

  b = [0 0 1]';

  G = I(3);
  a = [[ -1, 0, 1]
       [  0, 1, 0]
       [  0, 0, 1]];

  dx = A * x + b * u;
  dp = -G * (x - a);

  return [[dx][dp]];
}
```

となります。18 行目に注目して下さい。更新則を含めたすべての状態 [[dx][dp]] を返すわけで，これにともなって当然 link_eqs() のほうも書き換える必要があります。

今までは Ode() ですべての話しをしてきましたが，MATX はその目的に応じて多くの積分関数を持っています。以下にそれぞれの簡単なアルゴリズムと一緒に説明します。

8.3 rngkut4

4 次のルンゲクッタ法を行なう関数です。書式は以下の通りです。

```
Matrix rngkut4(Real t, Matrix x, Matrix dx(),
               Matrix u, Real dt)
```

t:	時間
x:	状態
dx():	dx を返す関数
u:	入力
dt:	きざみ時間

dt が省略された時には dt = 1.0E-7 が使われ，u が省略された時には，u = Z(0) が使われます。

これは for ループなどに用いることが出来ます。以下に例を示します。

```
Func void main()
{
  Integer i, ii;
  Real dt, t, t1, t2;
  Matrix x, u;
  Array X, U;
  Matrix plant(), controller();

  t1 = 0.0;  // 初期時間
  t2 = 5.0;  // 終端時間
  dt = 0.01; // 刻み時間
  ii = Integer((t2 - t1) / dt);

  x = [1  1  1]'; // 初期状態
  u = [0];

  X = Z(1, ii, x);
  U = Z(1, ii, u);

  t = t1;
  X(1, 1, x) = x;

  for (i = 1; i < ii; i++) {
    u = controller(t, x);
    x = rngkut4(t, x, plant, u, dt);
    U(1, i, u) = u;
    X(1, i+i, x) = x;
    t = t + dt;
  }

  print [[X][U]] >> "cont.mat";
}

Func Matrix plant(t, x, u)
  Real t;
  Matrix x, u;
{
  Matrix dx, A, b;
```

```
  A = [[ 0,  1,  0]
       [ 0,  0,  1]
       [-2, -3, -4]];

  b = [0 0 1]';

  dx = A * x + b * u;
  return dx;
}

Func Matrix controller(t, x)
  Real t;
  Matrix x;
{
  Matrix f, u;

  f = [ -1, -1, -1];
  u = f * x;
  return u;
}
```

このプログラムはシミュレーションの基礎となるプログラムです。 Ode() とはかなり形式が異なるので注意して下さい。 Ode() は dx を返す関数と，u を返す関数を作って，それを Ode() の引数に書けば，積分をやってその結果も書き出してくれますが，この rngkut4() は純粋にルンゲクッタ法をやってくれるだけなので，このように for で書く必要があります。ルンゲクッタ法の具体的なアルゴリズムは次のようになります。

$y' = f(x,y)$ という関数が存在し，(x_i, y_i) から y_{i+1} を求めるステップが次のようなアルゴリズムで示されているとき，$p+1$ 次のルンゲクッタ法であると言います。

$$
\begin{aligned}
y_{i+1} &= y_i + \Delta \mathrm{y} \quad (\mathrm{i} = 0, 1, 2, \ldots) \\
\Delta \mathrm{y} &= \lambda_0 k_0 + \lambda_1 k_1 + \cdots + \lambda_p k_p \\
k_0 &= hf(x_i, y_i) \\
k_1 &= hf(x_i + \alpha_1 h, y_i + \beta_1 k_0) \\
k_2 &= hf(x_i + \alpha_2 h, y_i + \beta_2 k_0 + \gamma_2 k_1) \\
\cdots &= \cdots \\
k_p &= hf(x_i + \alpha_p h, y_i + \beta_p k_0 + \gamma_p k_1 + \cdots)
\end{aligned}
$$

一般によく用いられる 4 次のルンゲクッタ法はこの係数を求めると，次のようになります。

$$
\begin{aligned}
y_{i+1} &= y_i + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3) \\
k_0 &= hf(x_i, y_i) \\
k_1 &= hf\left(x_i + \frac{h}{2}, y_i + \frac{k_0}{2}\right) \\
k_2 &= hf\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right) \\
k_3 &= hf(x_i + h, y_i + k_2)
\end{aligned}
$$

8.4 Ode()

この関数が MATX のシミュレーションの基本です。最初の例で解説しましたので，ここでは書式について説明します。

```
Ode(Real t0, Real t1, Matrix x,
    Matrix diff_eqs(), Matrix link_eqs(),
    Real h, real dtsav);

t0 :         初期時間
t1 :         終了時間
x  :         状態
diff_eqs :   dx を返す関数
link_eqs :   u を返す関数
h :          積分のきざみ幅
dtsav :      状態保存間隔
```

dtsav というのは Ode() が返してくる結果は先に述べたように {TC, XC, UC} となっています。 dtsav はこれらの積分の結果をセーブする時間を示します。きざみが小さくても，save はそれほど小さい必要はありません。 link_eqs() を省略した場合には入力が 0 となり，h が省略された時には，h = 4.0E-2 となり，dtsav が省略された時には，dtsav = 1.0E-3 となります。

使われているアルゴリズムは 4 次のルンゲクッタ法です。 for などを使わなくても，diff_eqs() と link_eqs() さえ作ればあとは全てやってくれるのでこの関数が全ての基本といってもいいでしょう。

ただし，きざみ幅には十分気を付けましょう。プラントによってはきざみ幅が大きいと安定な応答を示すが，きざみ幅を小さくすると不安定な応答を示すような場合があります。

数値積分においてきざみ幅というのは非常に重要なファクタとなります。プラントがゆっくり変化しているときにはきざみ幅は大きくても良いわけですし，急激に変化している時にはきざみ幅は小さくする必要が有ります。これらの話しは，8.6節に譲ります。

Ode() には数々のバリエーションがあります。以下にひとつづつ説明します。実際に用いる時にはこの巻末の表に従って下さい。

8.5 Hybrid

Hybrid がつくと，離散系のコントローラと考えて計算が行われます。書式は

```
OdeHybrid(Real t0, Real t1, Real dt, Matrix x,
          Matrix diff_eqs(), Matrix link_eqs(),
          Real h, Real dtsav)
```

dt がサンプリングタイムを示します。具体的には状態 x と時間 t から入力 u を返す関数 link_eqs() をサンプリングタイム毎に **1** 回ずつ呼び，サンプリングタイムの間はその入力を一定に保ちます。サンプリングタイムの時にしか link_eqs() を呼ばないというのが特徴です。離散系のコ

ントローラを用いるときには常にこれを用いる必要が有ります。

8.6　Auto

書式は以下のとおりです。

```
OdeAuto(Real t0, Real t1, Matrix x, Matrix dx(),
        Matrix input(), Real eps, Real dtsav)
```

eps は許容誤差を示します。省略すると eps = 1.0E-6 となります。 Auto がつくと，ルンゲクッタ法のきざみ幅を自動調節してくれます。 Ode() の説明で積分のきざみ幅の重要性について述べました。きざみ幅 h の値を小さくすれば，解の精度は上がります。逆に， h を大きくすれば概して精度は下がりますが，関数が急激に変化しないところでは h を大きめに取っても解の精度が落ちることは有りません。そこで，合理的に積分のきざみ幅を選ぶためには，**1** ステップを積分した時に生ずる誤差の評価を行なうことです。

きざみ幅の自動調節では， x を h 進める前に，次の 1, 2 を行なって h の値を決定します。

1. x を $h/2$ だけ 2 回進めたときの y の値を y_i^A として， x を h だけ 1 回進めたときの y の値を y_i^B とすると，

$$\frac{y_i^A - y_i^B}{y_i^A} \le \epsilon \quad (i = 0, 1, 2, \cdots)$$

の判定をします。 ϵ は制御定数で正の小さな値であり通常許容誤差を示します。この条件が満たされるまで h を $1/2$ ずつ小さくしていきます。

2. x を h だけ 2 回進めたときの y の値を y_i^C として， x を $2h$ だけ 1 回進めたときの y の値を y_i^D とすると，

$$\frac{y_i^C - y_i^D}{y_i^C} \le \epsilon \quad (i = 0, 1, 2, \cdots)$$

の判定をします。 ϵ は制御定数で正の小さな値であり通常許容誤差を示します。この条件が満たされるまで h を 2 倍ずつ大きくしていきます。

つまりきざみ幅を決定するためには積分 1 ステップの間にきざみ幅 h の計算とその半分ずつの計算の最低で 3 回の計算をすることになります。

積分結果が誤差以内でなかった場合はそのきざみを半分ずつ小さくするため時間がかかりますが，逆に関数の変化がゆるやかな場合は，きざみを大きくするので，全体の計算時間は短くなります。また，積分の誤差を完全に満たしているため信頼度は上がります。

8.7　RKF45

書式は以下の通りです。

```
rkf45(Real t, Matrix x, Matrix dx(), Matrix u,
      Real dt)
```

dt は省略すると dt=1.0E-7 となります。今まで述べてきた積分計算でルンゲクッタ法の 5 次の公式も導くことが出来ます。しかしこの場合は 1 ステップあたり関数計算を 1 回行なう必要が有ります。しかし Fehlberg によってこれが 6 回ですむことが試みられました。この方法を **RKF45** と呼びます。 5 次版のアルゴリズムは以下のようになります。

$$\begin{aligned}
k_1 &= hf(x_i, y_i) \\
k_2 &= hf(x_i + \frac{h}{4}, y_i + \frac{k_1}{4}) \\
k_3 &= hf\left(x_i + \frac{3h}{8}, y_i + \left(\frac{3k_1}{32} + \frac{9k_2}{32}\right)\right) \\
k_4 &= hf\left(x_i + \frac{12h}{13}, y_i + \left(\frac{1932k_1}{2197} - \frac{7200k_2}{2197} + \frac{7296k_3}{2197}\right)\right) \\
k_5 &= hf\left(x_i + h, y_i + \left(\frac{429k_1}{216} - 8k_2 + \frac{3680k_3}{513} - \frac{845k_4}{4104}\right)\right) \\
k_6 &= hf\left(x_i + \frac{h}{2}, y_i + \left(-\frac{8k_1}{27} + 2k_2\right.\right. \\
&\qquad \left.\left. - \frac{3544k_3}{2565} + \frac{1859k_4}{4104} - \frac{11k_5}{40}\right)\right) \\
\Delta y_i &= \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \\
y_{i+1} &= y_i + \Delta y_i
\end{aligned}$$

ここで更に 6 個の k_i のうち 4 個用いて y_{i+1} が 4 次まで正確になるように係数を選ぶことが出来， 4 次版のアルゴリズムは，

$$\Delta y_i = \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5$$

この方法は非常に計算回数を節約することが出来，しかも誤差は 5 次のルンゲクッタなので更に小さくすることが出来ます。 MATX では **rkf45** という関数がこのアルゴリズムを実現しています。

8.8　45Auto

Ode45Auto というのは前節の **RKF45** を用いてきざみ幅を自動調節する関数のことです。この計算ではこの **RKF45** の 4 次版と 5 次版を計算しその二つの誤差からきざみ幅を決定します。したがって誤差は 5 乗根で効いてくるので精度のよい積分計算を得ることが出来ます。 45Auto は最低で **1** ステップあたり **2** 回の計算を行なうので，**1** ステップあたり **3** 回計算する必要がある Auto よりは計算はかなり速いと言えます。

8.9 非線形プラント

今まで述べてきたプログラム例は全て線形プラントで，diff_eqs() の中の dx は行列計算で求まっていましたが，プラントが非線形であったり，コントローラに非線形性が入っていたとしても全く構いません。シミュレーションの基本を思い出して下さい。diff_eqs() は dx を返せばよいのですから，各成分毎にひとつひとつ微分方程式を書いておけば良いわけです。例えば，

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \cos(x_2) \\ \sin(x_1) \end{bmatrix}$$

という式を積分したければ，そのまま

```
dx(1,1) = cos(x(2,1))
dx(2,1) = sin(x(1,1))
```

と書くだけです。

8.10 積分のまとめ

以下に MATX に存在する全ての積分関数の書式を載せます。

```
List rngkut4(Real t, Matrix x, Matrix dx(),
        Matrix u, Real dt)
List Ode(Real t0, Real t1, Matrix x, Matrix dx(),
        Matrix input(), real h, Real dtsav)
List OdeAuto(Real t0, Real t1, Matrix x,
        Matrix dx(), Matrix input(),
        Real eps, Real dtsav)
List OdeHybrid(Real t0, Real t1, Real dt,
        Matrix x, Matrix dx(), Matrix input(),
        Real h, Real dtsav)
List OdeHybridAuto(Real t0, Real t1, Real dt,
        Matrix x, Matrix dx(), Matrix input(),
        Real eps, Real dtsav);
List rkf45(Real t, Matrix x, Matrix dx(),
        Matrix u, Real dt)
List Ode45(Real t0, Real t1, Matrix x,
        Matrix dx(), Matrix input(), Real h,
        Real dtsav)
List Ode45Auto(Real t0, Real t1, Matrix x,
        Matrix dx(), Matrix input(), Real eps,
        Real dtsav)
List Ode45Hybrid(Real t0, Real t1, Real dt,
        Matrix x, Matrix dx(), Matrix input(),
        Real h, Real dtsav)
List Ode45HybridAuto(Real t0, Real t1,
        Real dt, Matrix x, Matrix dx(),
        Matrix input(), Real eps, Real dtsav)
```

ではどの関数を使うべきでしょうか？

大きく分けて次の 2 種類に分けることが出来ます。

- 実験の前の実システムのシミュレーションの場合実験の前段階として，実際のプラントをモデリングして，シミュレーションを行なうならば，実験する時となるべく同じような状況設定が必要です。したがってサンプリング間隔で入力が一定となる Hybrid は必須です。
- 理論の確認の為の数値例としてのシミュレーションの場合理論の確認で連続系の理論ならば入力を一定にする必要はないので，Hybrid を使う必要はありません。

Hybrid を使う際に入力などの計算のために diff_eqs() の中に式を書くことが正しいかという点については，次のように考えることが出来ます。

link_eqs() の中でオイラー積分，あるいはルンゲクッタ法関数を使う積分を書く事と，diff_eqs() にその微分方程式を書いてしまう事の本質的な違いは，積分誤差をどこまで正確に見積もっているかどうかという点だけです。どちらに書いていても，Hybrid を使うことで入力が一定ということについてはシミュレーション出来るわけです。

したがって，速度的な面からも一番良いものは

```
Ode45HybridAuto()
```

でしょう。ただし，一概に断定できるものでもなく，全ての積分関数のアルゴリズムを理解して自分のシミュレーションに適切なものを選ぶことが重要なのは言うまでもありません。具体的な例については 11節で述べます。

9 計算結果の視覚化

シミュレーションが終ったらその結果を見る必要があります。

```
print [[TC][XC][UC]] >> "cont.mat"
```

でファイルに出力した結果は gnuplot でそのまま見ることができます。gnuplot の使い方はマニュアルに譲ります。

また，MATX から gnuplot を呼び出すことも出来ます。print のかわりに，

```
mgplot(1,TC,XC(1,:))
```

で状態 x1 を見ることが出来ます。複数のウィンドウにグラフを表示したり，1 個のウィンドウに複数のグラフを表示できますが，詳しくはマニュアルに譲ります。

また，xplot(古賀によって作られた)X-Window で動くプロットツールもあり，

```
% xplot cont.mat
```

という形で読み込むことが出来ます。GUI ツールなので説明は省きます。

10 H_∞ の為の設計

H_∞ の設計を MATX で行なうには，どうすればいいでしょうか？

MATX の H_∞ 設計関数 Hinf() は東京工業大学 制御システム工学科 寄附講座の助手の W.Manop さんが作られたものです。アルゴリズムは W.Manop さんの論文によっています。もちろん自分で H_∞ 問題を解く関数を作ってもか

まいません。基本的には 2 つのリカッチ方程式が解ければ良いわけですので，アルゴリズム的にはそれほど難しくはないといえます。 Hinf() はユーザ定義関数ですので，使い方は help で見ることが出来ます。 Hinf() の為の重み関数の設計などは BodePlot() を使うと良いでしょう。

11 2 軸ロボット Adaptive プログラム例

以下に 2 軸スカラロボットの Adaptive の例のプログラムを載せます。アルゴリズムは Slotine & Li(1986) によります。注意してほしいのですが，このプログラムはこのままでは走りません。計算のチェックも全くしていないので参考にする場合は計算が正しいかどうか自分で計算してから使うようにして下さい。例えば，33, 34 行目の lambda を始めとする諸定数はもちろん，35 行目の回帰行列 Y を求める関数も，目標値 qD, dqD, ddqD もありませんので，自分で作る必要があります。ここでは Adaptive のパラメータ更新則の積分は diff_eqs() の中で行なっており，diff_eqs() の返す dx は実プラントの状態と推定パラメータとで構成されています。

```
Func Matrix diff_eqs(t, x, u)
    Real t;
    Matrix x, u;
{
    Matrix M, C, tau;
    Matrix dx, q, dq;
    Real th1, th2, dth1, dth2;
    Matrix makeYcMatrix();

    M   = Z(2,2);    C   = Z(2,2);
    tau = Z(2,1);
    q   = Z(2,1);    dq  = Z(2,1);
    qr  = Z(2,1);    dqr = Z(2,1);

    th1   = x(1,1);    th2   = x(2,1);
    dth1  = x(3,1);    dth2  = x(4,1);
    tau = u;
    q  = [[th1][th2]];
    dq = [[dth1][dth2]];

    M(1,1) = a1 + a2 + 2 * a3 * cos(th2);
    M(1,2) = a2 + a3 *  cos(th2);
    M(2,1) = a2 + a3 *  cos(th2);
    M(2,2) = a2;

    C(1,1) = -a3 * sin(th2) * dth2 + a4;
    C(1,2) = -a3 * sin(th2) * (dth1 + dth2);
    C(2,1) =  a3 * sin(th2) * dth1;
    C(2,2) =  a5;

    ddq = D \ (tau - C * dq);

    ddth1 = ddq(1,1);
    ddth2 = ddq(2,1);

    dx = Z(9,1);

    dx(1,1) = dth1;
    dx(2,1) = dth2;
    dx(3,1) = ddth1;
    dx(4,1) = ddth2;  // プラント

    dqr = dqD  - lambda * (q  - qD);
    ddqr= ddqD - lambda * (dq - dqD);

    Yc = makeYcMatrix(dqr, ddqr, q, dq);

    s = dq - dqr;

    da_hat = - gamma * Yc' * s;

    dx(5:9,1) = da_hat;
    return dx;
}

Func Matrix link_eqs(t, x)
    Real t;
    Matrix x;
{
    Matrix a_hat;
    Matrix u, q, dq, qr, dqr;
    Real th1, th2, dth1, dth2;

    q   = Z(2,1);    dq  = Z(2,1);
    qr  = Z(2,1);    dqr = Z(2,1);

    th1   = x(1,1);    th2   = x(2,1);
    dth1  = x(3,1);    dth2  = x(4,1);

    q  = [[th1][th2]];
    dq = [[dth1][dth2]];

    a_hat = x(5:9,1);
    u = Z(2,1);

    dqr = dqD  - lambda * (q  - qD);
    ddqr= ddqD - lambda * (dq - dqD);

    Yc = makeYcMatrix(dqr, ddqr, q, dq);

    s = dq - dqr;

    u = Yc * a_hat - Kd * s;
    return u;
}
```

ここで賢明な人は，どうしてパラメータの更新則が diff_eqs() の中に書いてあるのだろう？これでは実際のインプリメントの時に実現できないのではないか？という疑問を持たれるでしょう。この疑問は正しいといえます。実際に計算機で制御プログラムを作る時にはコントローラは離散系ですから，このような積分は出来ないと言っても良いでしょう。

ある意味では，制御則は全て link_eqs() に書くべきだとも言えます。実際のインプリメントの時には，サンプリング時間が短いということで，大抵，パラメータ更新則の積分などをオイラー積分ですませることが多いようです (もちろんルンゲクッタ法関数 rngkut4() を入れるようにすればなお良いのですが)。ただし前述したように，その場合結局は diff_eqs() に書いた事と本質的な違いはなくなってしまいます。

したがって，シミュレーションの場合は積分を必要とするものは diff_eqs() の中に書く方が簡単になると思われます。このような理由で，この例プログラムはパラメータ

更新則を diff_eqs() の中に書いています。

12　おわりに

「論文を理解するには，よく読んで数式を追って証明を理解するのはもちろんですが，実際に論文で提案されている方法をシミュレーションしてみて初めて実感できるのではないでしょうか。」MATX を使うことはその意味で非常に有効だと思われます。

参考文献

シミュレーションの計算法については以下の本を参考にしています。

[1] 竹本宣弘, 荒 実, “C による数値計算”, 朝倉書店

[2] 奥村ほか監修, “Numerical Recipie in C”, 技術評論社