Maple による
科学技術計算のための可視化・プロット関連機能 - Part 2

サイバネットシステム株式会社

このワークシートでは、数式処理・数式モデル設計環境『Maple』を利用した、科学技術計算を行う際に便利な可視化・プロット機能について包括的な紹介をしています。Maple に用意されている豊富なプロット機能または関連機能を利用して、目的別の可視化を行ったり、また可視化・プロットに関連したデータ処理を用いて新しいアルゴリズムやアプリケーションの開発にお役立て頂けます。
なお、このワークシートはすでに Maple の基本的な操作方法について学習していることを前提にしています。Maple の基本的な操作方法については当社から提供されている「Maple ビギナーズガイド」等を参照してください。

目的と構成

Maple は数式処理に関する豊富な計算ツールを提供していますが、同時に数値計算に関する機能も豊富に提供しています。また、Maple は計算ツールとしてだけではなく、GUI 部品を用いた対話的なアプリケーション開発環境として、さらに最先端のマルチスレッド・並列計算等も用いたアルゴリズム開発環境としてもお使い頂けます。
このワークシートは、そのような科学技術計算全般で Maple をお使い頂けるようにするために、以下のように４つの構成から成っています。

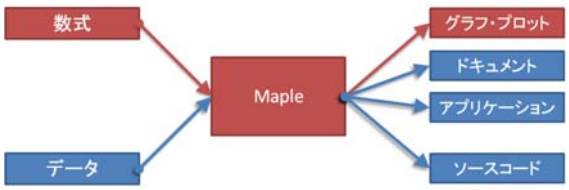
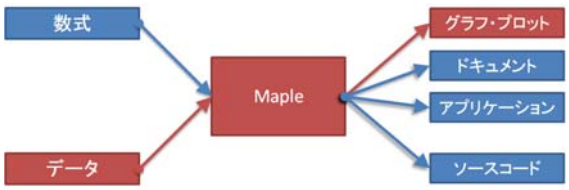
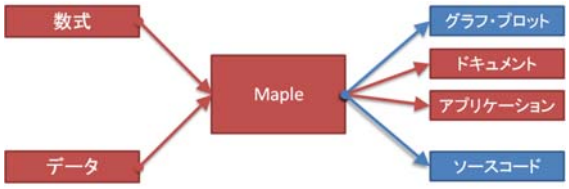
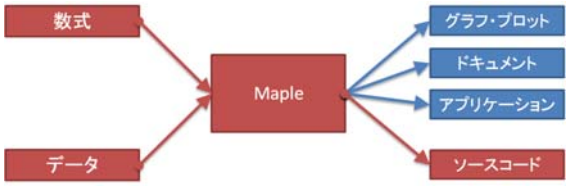
 利用例１：数式に関するグラフ・プロット作成	 利用例２：データに関するグラフ・プロット作成
 利用例３：ドキュメントやアプリケーション開発に関する関連機能	 利用例４：アルゴリズム開発のための関連機能

Table 1: 本ワークシートの構成

Part 1 では、利用例 1 ～ 2 までを紹介し、Part 2 では利用例 3 ～ 4 について紹介します。

関連リソース

Maple で利用可能な可視化・プロット機能の全体は、ヘルプ・コンテンツとして提供されている『プロットガイド』が便利です。

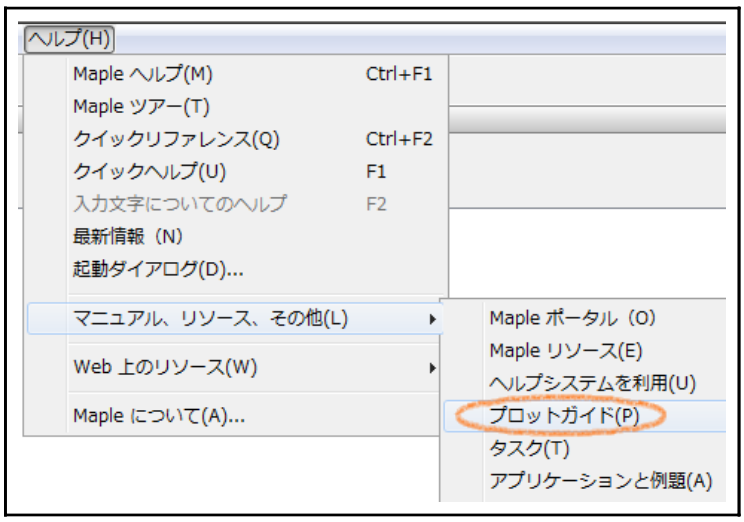


Table 2: 『プロットガイド』メニュー

『プロットガイド』は [ヘルプ] メニューから [マニュアル、リソース、その他] を選択し、『プロットガイド』のメニュー項目から選択可能です。

プロットガイドでは、目的別のプロットメニューへのリンクが掲載されています。等高線や密度線、ベクトル場の描画など目的別のプロットのヘルプへのリンクが用意されています。

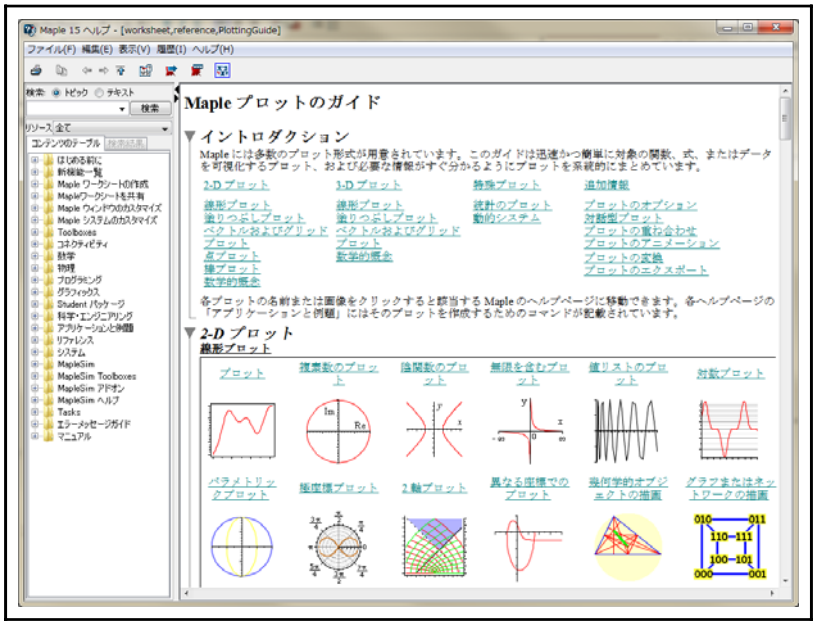


Table 3: プロットガイド

利用例 3 ：アプリケーション開発のための可視化・プロットと関連機能

GUI コンポーネントを用いたアプリケーション開発

Maple は計算コマンドやプログラミングを行なって計算・可視化をするための多くの機能を提供する統合的な科学技術計算環境ですが、これらの機能に加えて、Maple に詳しくない人でも計算やプ

ロットを手軽に実現するための GUI コンポーネントを Maple ワークシートに配置する機能を用意しています。
GUI コンポーネントを用いたワークシートでは、ユーザは通常の Windows/MacOS X アプリケーションなどと同様に、スライダーやテキストボックスの操作を行うだけで計算・プロットを行うことが可能です。

ここでは、そのようなアプリケーションを開発するための基本的な機能について紹介します。

GUI 部品は、Maple ワークシート・ウィンドウ左側にある「パレット」領域内にある「コンポーネント」パレットからワークシート上へ配置します。

Table 4: GUI コンポーネント・パレット

また、各コンポーネントを適切な場所に配置するために、[挿入]メニューから[テーブル]を選択してレイアウトを行うことも重要です。

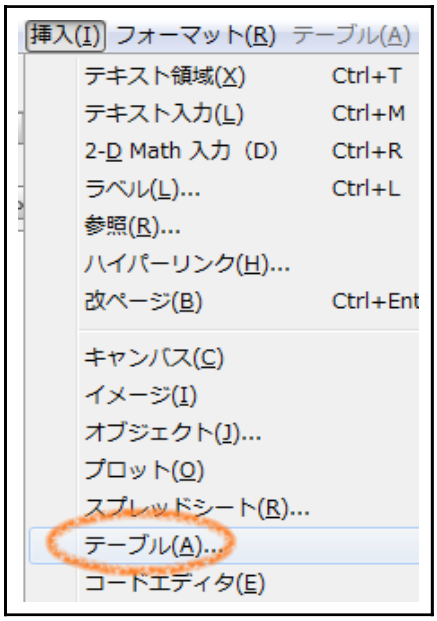
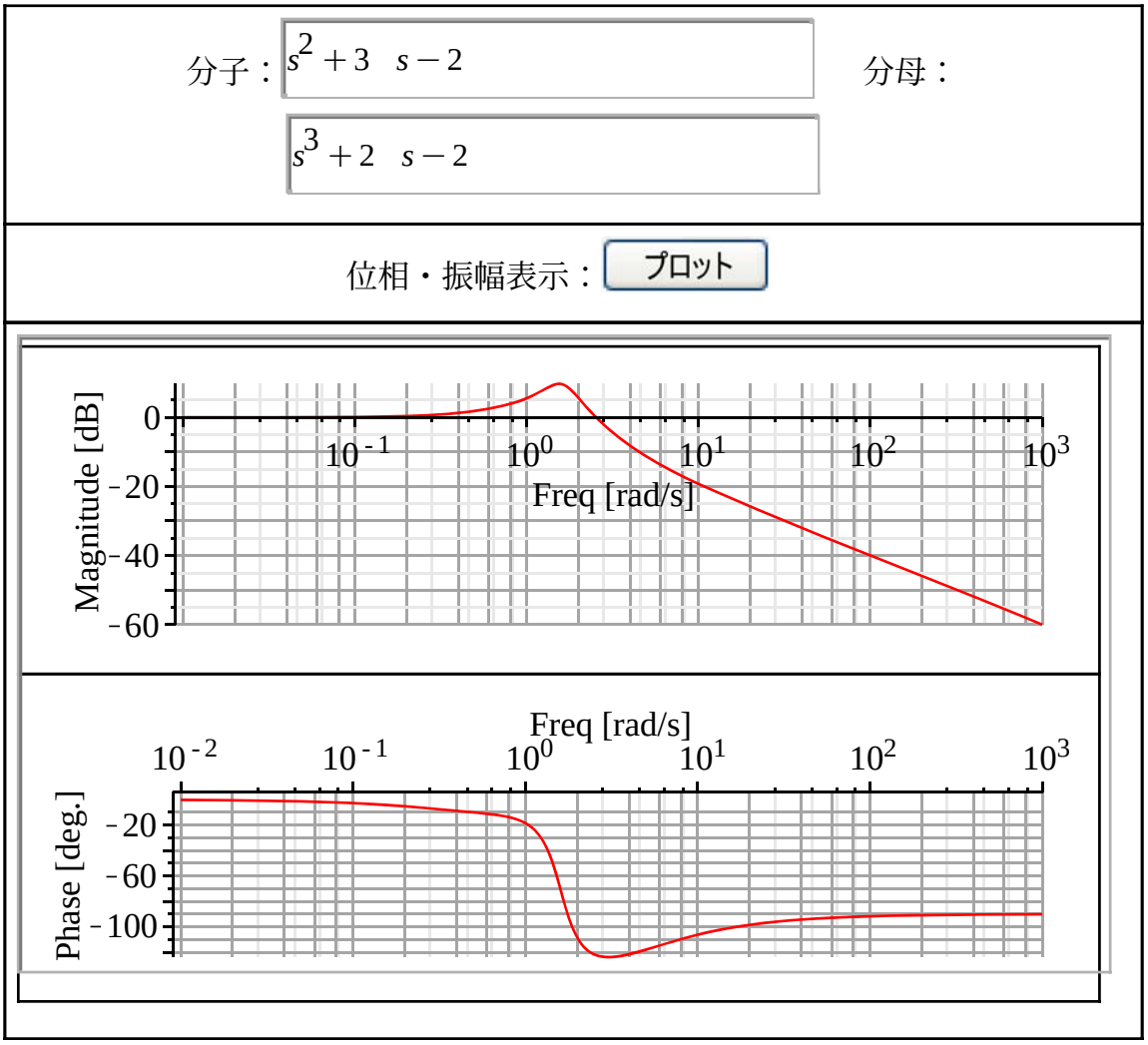


Table 5: テーブル メニュー

ここでは、次のようなレイアウトに各 GUI コンポーネントを配置した構成を考えます。これは、ある伝達関数（分子・分母の式を指定）し、その伝達関数の位相・振幅をグラフ表示するものです。



アプリケーションのコード

このアプリケーションで必要なコードは、以下の通りです：
Step1: 分子・分母の数式を取得し、伝達関数を構成する
Step2: 用意した伝達関数から位相・振幅のプロットを作成しプロットウィンドウに表示する

それぞれのコードを見てみます。
Step1: 分子・分母の数式を取得し、伝達関数を構成する

```
> restart;
> # 数式を取得する関数を定義
  getExpression := proc()
    uses DocumentTools; # このプロシージャ内部で使うパッケージを宣言
    local snumer, sdenom;
    snumer := Do(%mcNumer); # 分子
    sdenom := Do(%mcDenom); # 分母
    return (snumer/sdenom); # 分数式にして返す
  end proc;
```

```
getExpression := proc( )                                    (3.1.1.1)
    local snumer, sdenom;
    snumer := DocumentTools:-Do(%mcNumer);
    sdenom := DocumentTools:-Do(%mcDenom);
    return snumer / sdenom
end proc
```

```
> # テストしてみます。
  getExpression();
```

$$\frac{s^2+3s-2}{s^3+2s-2} \qquad \textbf{(3.1.1.2)}$$

```
> # 取得した式から DynamicSystems パッケージの
  # 伝達関数オブジェクトを構成する
  makeTF := proc()
    uses DynamicSystems;
    TransferFunction(getExpression());
  end proc;
```

makeTF := **proc**( ) *DynamicSystems*:-*TransferFunction*(*getExpression*( )) **end proc** **(3.1.1.3)**

```
> # テストしてみます。
  DynamicSystems[PrintSystem](makeTF());
```

Transfer Function

continuous

1 output(s); 1 input(s)

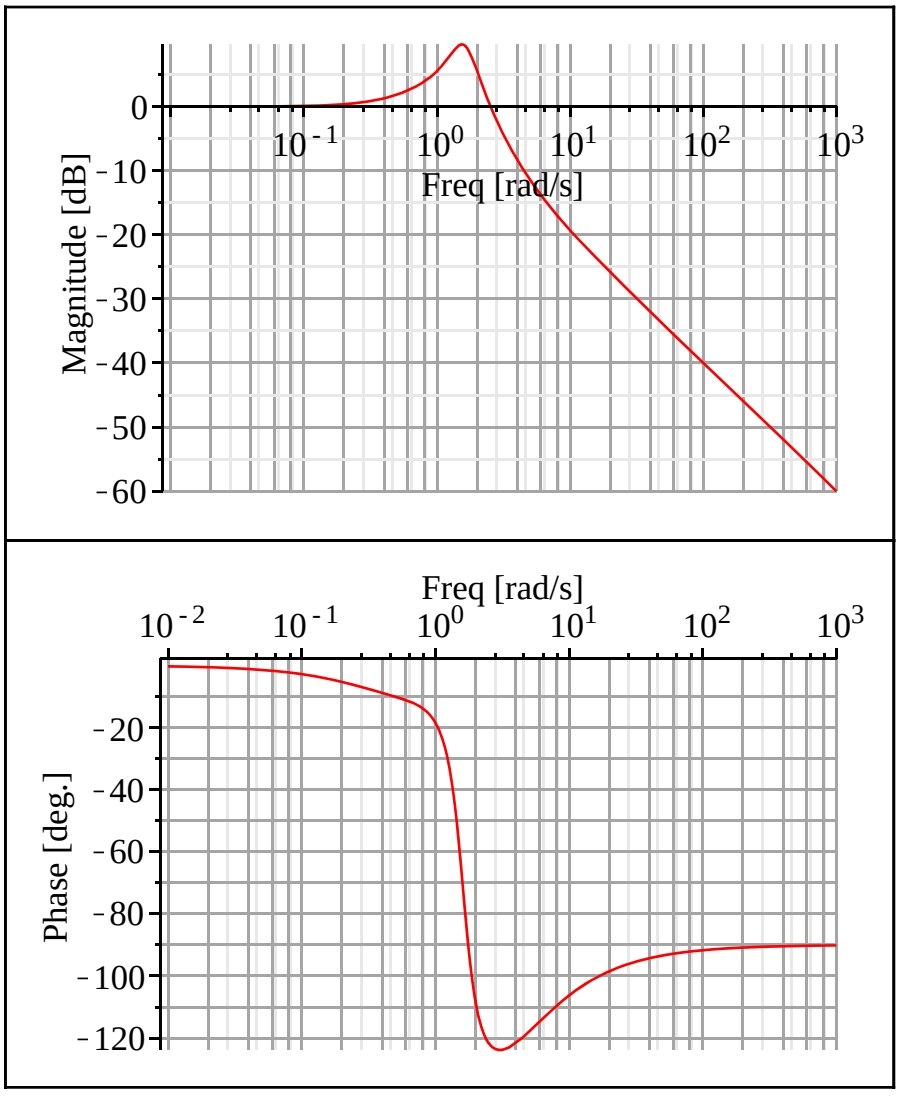
inputvariable = [*u1*(*s*)]

outputvariable = [*y1*(*s*)]

$$tf_{1,1} = \frac{s^2+3s-2}{s^3+2s-2} \qquad \textbf{(3.1.1.4)}$$

Step2: 用意した伝達関数から位相・振幅のプロットを作成しプロットウィンドウに表示する

```
> # BodePlot コマンドで位相・振幅プロットを作成できることを確認
  DynamicSystems[BodePlot](makeTF());
```



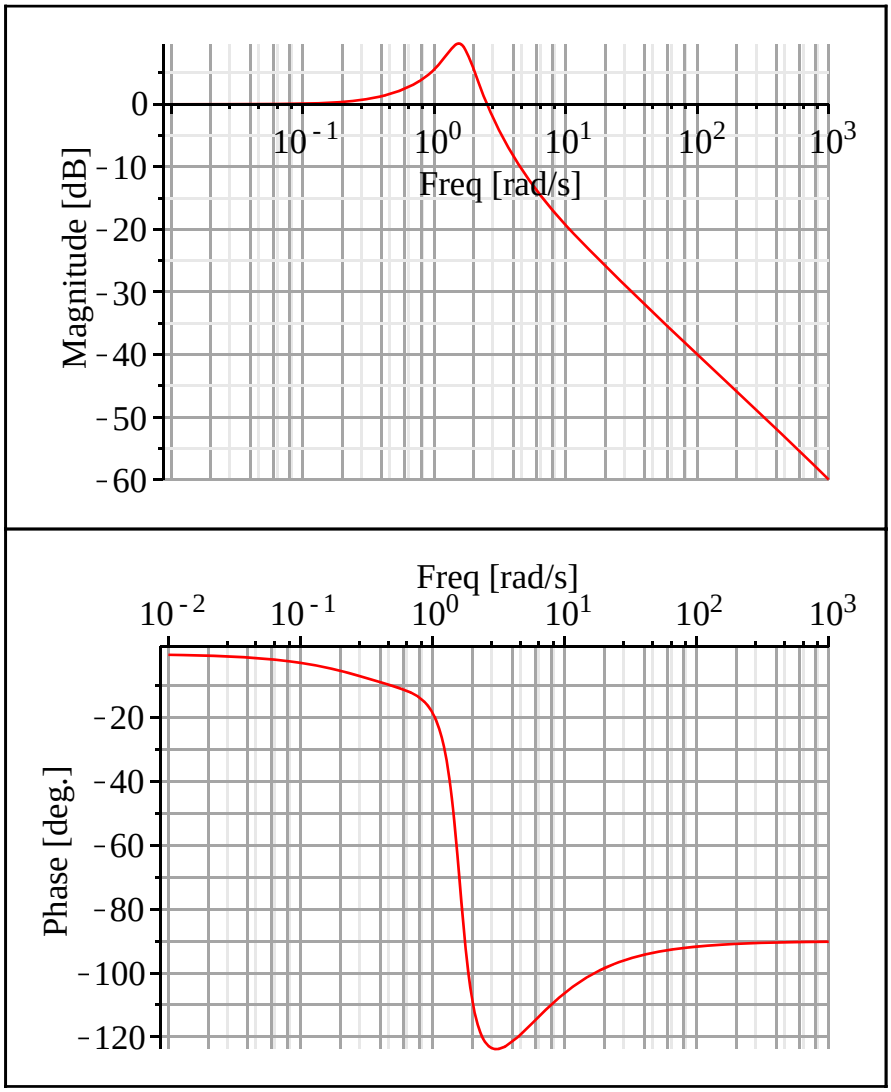
```
> # 振幅・位相のプロットをプロットウィンドウにセットする関数を定義
  setTFBodePlot := proc()
    uses DocumentTools, DynamicSystems;
    Do(%Plot0 = BodePlot(makeTF()));
  end proc;
```

setTFBodePlot := **proc**( ) **(3.1.1.5)**

DocumentTools:-*Do*(*%Plot0* = *DynamicSystems*:-*BodePlot*(*makeTF*( )))

end proc

```
> # テストしてみます。
> setTFBodePlot();
```



以上でコードの作成は終了です。最後に用意した setTFBodePlot 関数をテキストボタンの [クリックしたときの動作] に記述します。

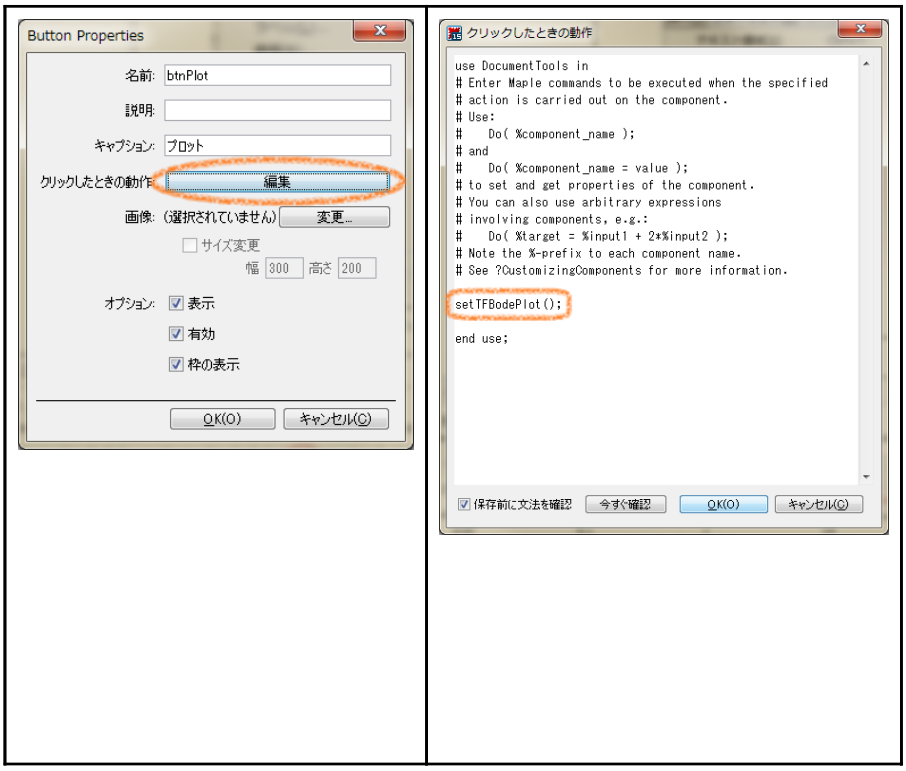


Table 6: ボタン押下時のアクション指定

なお、作成したコードは、いずれも一度実行しなければ正しく動作しません。(カーネル初期化時にはその定義は消滅します)
アプリケーション読込時やカーネルの再起動時に、ユーザ定義部分のコードを自動的に実行するには、以下のどちらかを実施しておかなければなりません。

1. [編集]メニューから[スタートアップコード] を選択し必要な関数定義を記述しておく。スタートアップコードはワークシート内で使われる関数を定義するための領域です。ワークシートが開かれた際に実行されます。

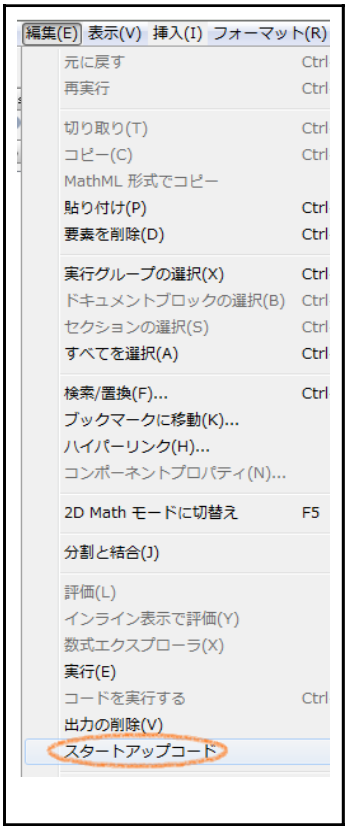


Table 7: スタートアップコード

2. ユーザ定義関数を含むドキュメントブロックの [自動実行] プロパティを設定しておく。

ドキュメントブロックの取扱方法の詳細については、ヘルプドキュメント、または Maple ビギナーズガイドを参照してください。

アプリケーションの配布形態

Maple ワークシートと GUI コンポーネントを用いたアプリケーションは、以下の３つの方法で配布することが可能です。

1. Maple ワークシートとして配布する
作成したワークシートを、Maple 本体を持つ他のユーザにそのまま渡すだけで実行可能なアプリケーションとして動作します。

2. Maple Player 用コンテンツにする
GUI コンポーネントを用いたワークシート型アプリケーションは、Maple Player (閲覧と対話操作を実行可能なビューワ)で実行するためのアプリケーションにして配布することが可能です。
Maple Player 用コンテンツ(拡張子：.mwz)への変換には Maplesoft 技術サポートへお問い合わせください。

3. MapleNET コンテンツとして配布する
MapleNET は、Maple の計算エンジンをウェブ環境で利用するためのサーバ製品です。MapleNET の指定フォルダへ作成したコンテンツを保存するだけで、GUI コンポーネントを含む対話型Webアプリケーションとしてご利用頂けます。

利用例４：アルゴリズム開発のための関連機能

Maple 上での並列処理・マルチスレッド処理

近年の PC では、マルチコア CPU を利用する機会が増えてきています。Maple はマルチコア CPU やグリッド環境を想定した計算アプリケーションのためのアルゴリズムを開発するプロトタイプ環境としてもお使い頂けます。

なお、Maple での並列プログラミングの詳細については、Maple プログラミングガイド 14 章を参照してください。

並列計算は、データ処理を必要とする多くのアプリケーションで計算速度の観点でメリットがある場合があります。
以下は、マンデルブロ集合を描画するための２種類のコード実装例です。
（詳細はプログラミングガイドを参照してください）
なお、本節の例題を実行する際はマルチコア CPU 環境であり、かつ十分なメモリ量が搭載されていることを確認下さい。

１番目の実装例：
次のコードエディタ内に記述されているのは、シングルスレッドのみでマンデルブロ集合を計算するコードです。

```
> restart;
```

```
Mandelbrot := module()
```

注：コードエディタ内のコードを実行するには、コードエディタボタンをそのままクリックするか、または右クリックから表示されるコンテキストメニューで [コードエディタを展開]を選択し、再び右クリックから [コードを実行] を選択します。

コードエディタのコードを実行したら、次の Maple コードを実行してシングルスレッドで実行した場合の計算時間を計測してみます。

```
> # サンプル点を500個として計算
```

```
N := 500:
s := time[real]():
points := Mandelbrot(N, N, 100, -2.0, .7, -1.35, 1.35, 10.0):
time[real]() - s;
```

50.014 **(4.1.1)**

２番目の実装例：
次のコードエディタ内に記述されているのは、このワークシートが実行されている Maple の PC のマルチコアをすべて利用し、かつクライアント・サーバ方式で複数 CPU コアをフルに活用して計算を実行するコードです。

```
> restart;
```

```
csgridMandelbrot := module()
```

コードエディタのコードを実行したら、先と同様に次のコードを実行して計算時間を計測します。

```
> # サンプル点を500個として計算
  N := 500:
  s := time[real]():
  points := csgridMandelbrot(N, N, 100, -2.0, .7, -1.35, 1.35, 10.0):
  time[real]() - s;
```

13.338 **(4.1.2)**

参考までに、以下は Intel Core i7 920 (8 core) 2.67GHz, 3.0GB RAM, Windows 7 (64bit) 上の Maple 15 (64bit) で実行した結果です。
シングルスレッドの計算と比較して３倍程度の高速化が図れています。

Table 8: 並列計算によるパフォーマンス

Maple による言語変換（コード生成）

Maple は数式処理・数値計算の統合計算環境ですが、その数式処理の応用により計算量を最適化した他のプログラミング言語のコードの生成のためにお使い頂くことも可能です。

Maple に標準で提供されている CodeGeneration パッケージは、Maple 上で定義されたプログラムコードを複数の言語のコードへと変換するための機能を提供しています。
注：Maple の組込みの計算関数・コマンド、例えば記号式に対する微分・積分演算、微分方程式を記号的に解く、等の関数・コマンドは他の言語のコードに変換することが出来ません。

Maple の機能を利用した、計算量の解析や式の書換えによる簡単化などの例については、リンクされている適用事例などを参照してください。
このワークシートでは、簡単に CodeGeneration パッケージの基本機能について紹介します。

```
> # CodeGeneration パッケージの読込み
  restart;
  with(CodeGeneration);
```

[*C, CSharp, Fortran, IntermediateCode, Java, LanguageDefinition, Matlab, Names, Save, Translate, VisualBasic*] **(4.2.1)**

```
> # 以下はユーザ定義関数の一例です。
  # この関数は xin の1次元配列データに対して、要素の二乗和を計算し
  # その平方根で各要素を標準化（スケーリング）しています。
  # xin は入力用配列、xout は出力用配列、n は配列の長さです。
  myfunc := proc(xin::Array(datatype=float[8]), xout::Array(datatype=
  float[8]), n::posint)
    local i::integer, stemp::float;

    stemp := 0.0;
    for i from 1 to n do
      stemp := stemp + xin[i]^2;
    end do;

    stemp := sqrt(stemp);
    for i from 1 to n do
      xout[i] := xin[i]/stemp;
    end do;

  end proc:
> # 定義した myfunc を C言語のコードに変換
  C(myfunc);
#include <math.h>

double myfunc (double *xin, double *xout, int n)
{
  int i;
  double stemp;
  double cgret;
  stemp = 0.0e0;
  for (i = 1; i <= n; i++)
    stemp = stemp + pow(xin[i - 1], 0.2e1);
  stemp = sqrt(stemp);
  for (i = 1; i <= n; i++)
  {
    xout[i - 1] = xin[i - 1] / stemp;
    cgret = xout[i - 1];
  }
  return(cgret);
}
> # 定義した myfunc を FORTRAN コードに変換
  Fortran(myfunc);
      doubleprecision function myfunc (xin, xout, n)
        doubleprecision xin(*)
        doubleprecision xout(*)
        integer n
        integer i
        doubleprecision stemp
        doubleprecision cgret
        stemp = 0.0D0
        do 100, i = 1, n, 1
          stemp = stemp + xin(i) ** 2
100     continue
        stemp = sqrt(stemp)
        do 200, i = 1, n, 1
          xout(i) = xin(i) / stemp
          cgret = xout(i)
200     continue
        myfunc = cgret
        return
      end
> # 定義した myfunc を Java コードに変換
  Java(myfunc);
import java.lang.Math;

class CodeGenerationClass {
  public static double myfunc (double[] xin, double[] xout, int n)
  {
    int i;
    double stemp;
    double cgret;
    stemp = 0.0e0;
```

```
    for (i = 1; i <= n; i++)
      stemp = stemp + Math.pow(xin[i - 1], 0.2e1);
    stemp = Math.sqrt(stemp);
    for (i = 1; i <= n; i++)
    {
      xout[i - 1] = xin[i - 1] / stemp;
      cgret = xout[i - 1];
    }
    return(cgret);
  }
}
```

同様にして、C# や Visual Basic, MATLAB 等へのコードにも変換が可能です。

ここで定義した myfunc 関数はそのまま Maple 上で利用することもできますが、より高速に実行するために、Compiler パッケージを利用した Maple 上のユーザ定義関数の DLL 化を検討する価値があります。（Maple の基本的かつ数値的な演算のみによるユーザ定義関数の場合、Compiler パッケージによる DLL 化で高速計算が実現できる場合があります）

```
> # 配列長を用意
  N := 10^6;
```

$$N := 1000000 \tag{4.2.2}$$

```
> # xin, xout のための配列を用意
  # myfunc 関数の定義により Array に datatype=float[8] オプションを
  # 指定していることに注意。これは 8byte 浮動小数型を意味しています。
  inData := Array(1..N,i->i/(i+1),datatype=float[8]):
  outData1 := Array(1..N,datatype=float[8]):
  outData2 := Array(1..N,datatype=float[8]):
> # Maple 上で計算した場合の計算時間・メモリ使用量を計測
  CodeTools:-Usage(myfunc(inData,outData1,N)):
memory used=175.80MiB, alloc change=0 bytes, cpu time=3.24s, real time=3.26s
> # myfunc 関数を DLL 化し、計算時間・メモリ使用量を計測
  cpmyfunc := Compiler:-Compile(myfunc):
> CodeTools:-Usage(cpmyfunc(inData,outData2,N)):
memory used=0.63KiB, alloc change=0 bytes, cpu time=31.00ms, real time=30.00ms
> # 計算した2つのデータ結果の先頭5要素を確認
  outData1[1..5],outData2[1..5];
```

[0.000500006535258721, 0.000666675380344962, 0.000750009802888082, 0.000800010456413954, 0.000833344225431202], [0.000500006535258721, 0.000666675380344962, 0.000750009802888082, 0.000800010456413954, 0.000833344225431202] **(4.2.3)**