unapply コマンドを用いた
ニュートン法の簡単なコーディング

unapply コマンドは Maple に用意されている非常に便利なコマンドです。このコマンドを用いることで、通常は冗長になりがちなプログラム作成も非常に簡潔に記述することが可能です。また、数式処理の特徴を生かして、ニュートン法の反復で用いられる式を明示的に得ることもできます。
この資料では、unapply コマンドの使い方を解説すると共に、例題として数値解法の単変数・多変数版ニュートン法のコードの書き方を紹介します。

[注意] 本資料はプログラミングに関連した話題のため、ワークシート中の数式及びコマンドは、テキストモードで記述しています。数式モードで記述しても基本的には問題ありません。

unapply コマンドとは？

unapply コマンドは、与えられた引数に関する関数オペレータを返すためのコマンドです。端的に言えば、式を与えて関数を返すためのコマンドとなります。
例えば、unapply コマンドに sin(x) を与えてみます。

```
> restart;
> unapply(sin(x),x);
```

$$x \mapsto \sin(x) \tag{1.1}$$

戻り値は、矢印→で示されるように関数となります。
戻り値を別の変数に割り当てると、それ自体がユーザ定義の関数として使うことが可能です。

```
> f := unapply(sin(x),x);
```

$$f := x \mapsto \sin(x) \tag{1.2}$$

```
> f(x);
```

$$\sin(x) \tag{1.3}$$

```
> f(1.2);
```

$$0.9320390860 \tag{1.4}$$

unapply コマンドは、ある計算を行った結果を関数として利用する場合に便利なコマンドです。
次は、式 $\sin(x)\cdot\cos(x)$ の微分を計算し、与えられた点での微係数を返すための記述例です。

```
> df := unapply(diff(sin(x)*cos(x),x),x);
```

$$df := x \mapsto \cos(x)^2 - \sin(x)^2 \tag{1.5}$$

```
> df(1.0);
```

$$-0.4161468365 \tag{1.6}$$

```
> df(Pi);
```

$$1 \tag{1.7}$$

通常、Maple ではユーザ定義関数を → で記述しますが、上記の例を次のように記述すると一見正しく定義できているように思われますが、計算を行うとエラーとなります。

```
> df2 := x -> diff(sin(x)*cos(x),x);
```

$$df2 := x \to \frac{d}{dx}\left(\sin(x)\cos(x)\right) \tag{1.8}$$

```
> df2(3);
Error, (in df2) invalid input: diff received 3, which is not valid for its 2nd argument
```

df2 の定義では、引数x に対して右辺部分の計算を後から行う形になります。すると、

```
> diff(sin(3)*cos(3),3);
```
Error, invalid input: diff received 3, which is not valid for its 2nd argument

という形で式が評価され、エラーとなってしまいます。

unapply によるニュートン法コード：単変数の場合

ニュートン法とは、$f(x)$ の零点を次の反復式によって求める算法です；

$$x_{n+1} = x_n - \frac{f(x_n)}{\left.\frac{\mathrm{d}}{\mathrm{d}x} f(x)\right|_{x=x_n}}$$

ここで、$x = x_0$を初期値としています。

数式処理である Maple では、式の微分を diff コマンドにより明示的に計算することが可能です。従って、Maple ではニュートン法をアルゴリズムの記述に近い形でプログラム化することが可能です。

まず、ニュートン法の反復のメイン部分である上記式のプロシージャを定義してみます。戻り値を unapply コマンドにより得ていることに注意していください。

```
> makeIteration := proc(f,v)
    local ef,fret;

    ef := evalf(f); # f の中に含まれる数学定数含めて数値化
    fret := v - ef/diff(ef,v);
    unapply(fret,v);
  end proc;
```

makeIteration := **proc**(*f*, *v*) **(2.1)**

local *ef*, *fret*;

ef := *evalf* (*f*); *fret* := *v* − *ef* / *diff* (*ef*, *v*); *unapply*(*fret*, *v*)

end proc

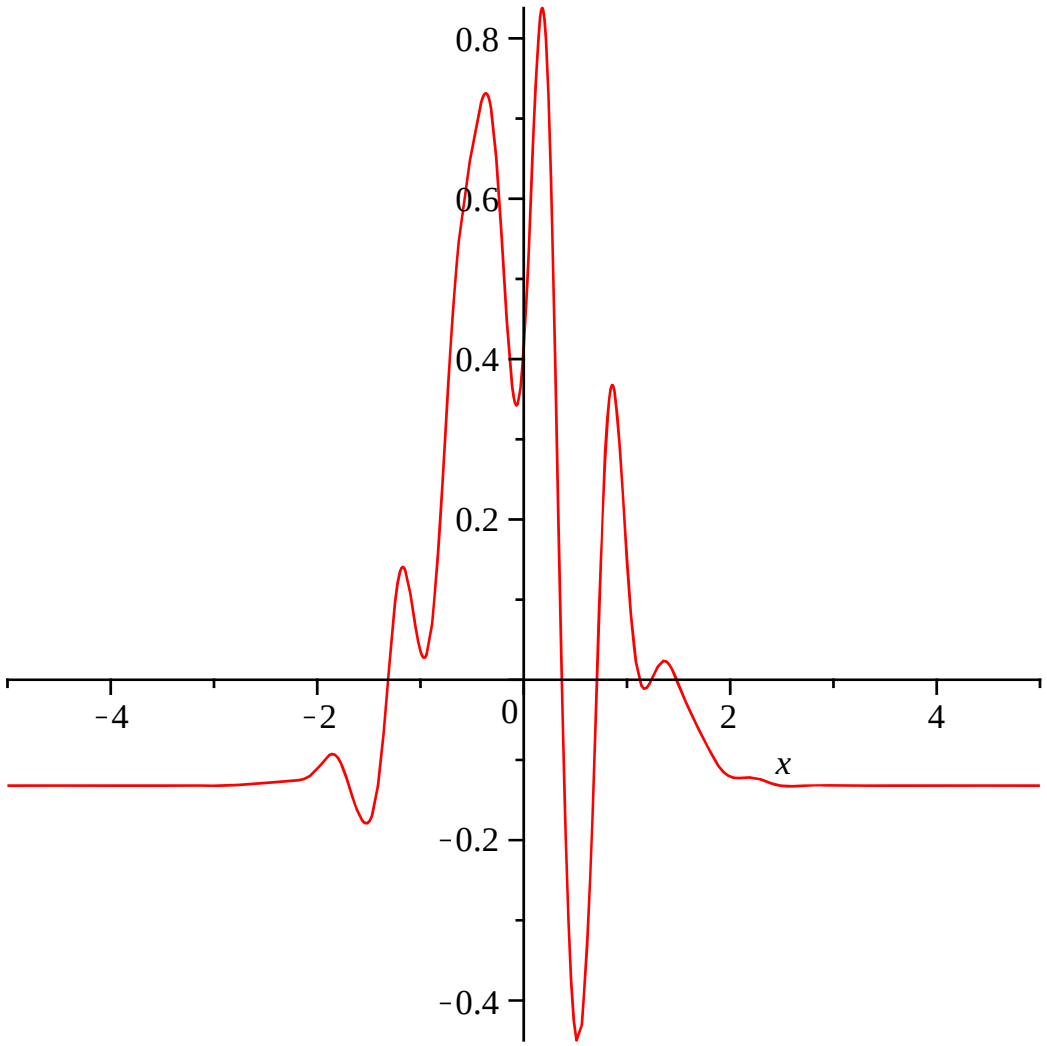
makeIteration 関数を実際の問題で試してみます。次の問題を考えます；

```
> f := exp(-0.918*x^2)*cos(cos(6.032*(x-Pi))-sin(2.992*x))
  -0.132;
```

$$f := \mathrm{e}^{-0.918\,x^2}\cos\left(\cos\left(6.032\,x - 6.032\,\pi\right) - \sin\left(2.992\,x\right)\right) - 0.132 \qquad \textbf{(2.2)}$$

この関数 f を図示してみます。

```
> plot(f,x=-5..5);
```



makeIteration 関数を適用することで、関数 f から反復を計算するための関数を得ます。

```
> newton_uni := makeIteration(f,x);
```

$$newton\_uni := x \mapsto x - \left(e^{-0.918\,x^2}\cos(\cos(6.032\,x - 18.95008689) - \sin(2.992\,x)) - 0.132\right) \Big/ \left(-1.836\,x\,e^{-0.918\,x^2}\cos(\cos(6.032\,x - 18.95008689) - \sin(2.992\,x)) - e^{-0.918\,x^2}\sin(\cos(6.032\,x - 18.95008689) - \sin(2.992\,x))\,(-6.032\sin(6.032\,x - 18.95008689) - 2.992\cos(2.992\,x))\right) \quad \textbf{(2.3)}$$

後は、適当な初期値から反復計算を行います（ここでは20回反復させています）；

```
> x0 := 0.5;
  to 20 do
    x0 := newton_uni(x0);
  end do;
```

$$x0 := 0.5$$
$$x0 := -0.0091539586$$
$$x0 := -0.2344337576$$
$$x0 := 0.0657064221$$
$$x0 := -0.1220688959$$
$$x0 := 0.1673553517$$
$$x0 := -1.080521449$$
$$x0 := -0.9687678513$$

$$x0 := -0.5623409583$$
$$x0 := -1.292325340$$
$$x0 := -1.313274471$$
$$x0 := -1.312904892$$
$$x0 := -1.312904895$$
$$x0 := -1.312904894$$
$$x0 := -1.312904895$$
$$x0 := -1.312904894$$
$$x0 := -1.312904895$$
$$x0 := -1.312904894$$
$$x0 := -1.312904895$$
$$x0 := -1.312904894$$
$$x0 := -1.312904895 \quad \textbf{(2.4)}$$

得られた値を f に代入してみます。

```
> evalf(eval(f,x=x0));
```

$$-1.1\ 10^{-9} \quad \textbf{(2.5)}$$

別の初期値からも零点を探してみます。なお、ここでは反復過程中に精度のチェックを行ってみます。$\mathrm{abs}(x_i - x_{i-1}) < 10^{-Digits}$ であれば反復を終了するようにします。

```
> x1 := 1.5;
  to 20 do
    tmp := x1;
    x1 := newton_uni(x1);
    print(x1);
    if abs(x1-tmp)<10.0^(-Digits) then break; end if;
  end do:
```

$$x1 := 1.5$$
$$1.482378298$$
$$1.482197202$$
$$1.482197166$$
$$1.482197166 \quad \textbf{(2.6)}$$

得られた結果を検算しておきます。確かに（精度内で）十分ゼロに近い値となっていることがわかります。

```
> evalf(eval(f,x=x1));
```

$$1.\ 10^{-10} \quad \textbf{(2.7)}$$

この資料で定義した makeInteration 関数と反復計算を組み合わせて、ユーザ独自のニュートン法を定義する場合は、以下のような形で記述できます。

```
> NewtonIteration := proc(f,var,ini,ite)
    # f: function, var: variable, ini: initial value, ite: num
  of iteration
    local nf, x0, tmp;

    nf := makeIteration(f,var);
    x0 := ini;
    to ite do
      tmp := x0;
      x0 := nf(x0);
      if abs(x0-tmp)<10.0^(-Digits) then break; end if;
    end do;
    return(x0);

  end proc;
```

$$NewtonIteration := \mathbf{proc}(f, var, ini, ite) \quad \textbf{(2.8)}$$

```
    local nf, x0, tmp;
    nf := makeIteration( f, var);
    x0 := ini;
    to ite do
        tmp := x0; x0 := nf (x0); if abs(x0 − tmp) < 10.0^( −Digits) then break
        end if
    end do;
    return x0
end proc
```

テストしてみましょう。$x = 0.1$ を初期値として、零点を計算してみます。

```
> NewtonIteration(f, x, 0.1, 100);
```

0.0465195448 **(2.9)**

複数の（ランダムな）初期値から計算する場合は、seq コマンドを用いて次のように記述します。ここでは5個の零点を求めています。

```
> seq(NewtonIteration(f,x,rand()/10.0^12,100), k=1..5);
```

0.3691703237, 1.237760534, −1.312904894, 0.7095117909, 0.3691703241 **(2.10)**

unapply によるニュートン法コード：多変数の場合

前節で定義した単変数のニュートン法のコードは、少し修正を加えるだけで多変数に拡張することが可能です。この場合も unapply コマンドを使うことで非常に簡潔にコードを記述することができます。

次式は、ニュートン法を多変数の場合に拡張したものです；

$$X_{n+1} = X_n - J(X_n)^{-1} \cdot f(X_n)$$

ここで、X: 変数ベクトル、X_n: n 番目の計算結果、$f(X)$: 非線形方程式系で、$J(X_n)$ は $X = X_n$ におけるヤコビアンです。

本質的には単変数の場合とほとんど同一的な記述で多変数用のニュートン法のプログラムも作成できます。まず、ヤコビアンの計算方法について紹介します。

VectorCalculus パッケージを用いる場合

VectorCalculus パッケージにはヤコビアンを計算するための Jacobian コマンドが用意されています。

```
> with(VectorCalculus);
```

[&x, `*`, `+`, `-`, `.`, <,>, <|>, About, AddCoordinates, ArcLength, BasisFormat, Binormal, Compatibility, ConvertVector, CrossProd, CrossProduct, Curl, Curvature, D, Del, DirectionalDiff, Divergence, DotProd, DotProduct, Flux, GetCoordinateParameters, GetCoordinates, GetNames, GetPVDescription, GetRootPoint, GetSpace, Gradient, Hessian, IsPositionVector, IsRootedVector, IsVectorField, Jacobian, Laplacian, LineInt, MapToBasis,∇, Norm, Normalize, PathInt, PlotPositionVector, PlotVector, PositionVector, PrincipalNormal, RadiusOfCurvature, RootedVector, ScalarPotential, SetCoordinateParameters, SetCoordinates, SpaceCurve, SurfaceInt, TNBFrame, Tangent, TangentLine, TangentPlane, TangentVector, Torsion, Vector, VectorField, VectorPotential, VectorSpace, Wronskian, diff, eval, evalVF, int, limit, series] **(3.1.1)**

```
> Jacobian([r*cos(t),r*sin(t),r*t^2], [r,t]);
```

$$\begin{bmatrix} \cos(t) & -r\sin(t) \\ \sin(t) & r\cos(t) \\ t^2 & 2rt \end{bmatrix} \quad (3.1.2)$$

```
> unwith(VectorCalculus):
```

自身でヤコビアンを定義する場合

for-loop 文を用いてヤコビアンを計算するプログラムを簡単に記述できます。

```
> jacob := proc(fl::list, vl::list)
    local i,j,J;

    J := Matrix(nops(fl),nops(vl));
    for i to nops(fl) do
      for j to nops(vl) do
        J[i,j] := diff(fl[i],vl[j]);
      end do;
    end do;
    return(J);
  end proc;
```

jacob := **proc**(*fl*::*list*, *vl*::*list*) (3.2.1)

local *i*, *j*, *J*;

J := *Matrix*(*nops*(*fl*), *nops*(*vl*));

for *i* **to** *nops*(*fl*) **do**

for *j* **to** *nops*(*vl*) **do** *J*[*i*, *j*] := *diff* (*fl*[*i*], *vl*[*j*]) **end do**

end do;

return *J*

end proc

```
> jacob([r*cos(t),r*sin(t),r*t^2],[r,t]);
```

$$\begin{bmatrix} \cos(t) & -r\sin(t) \\ \sin(t) & r\cos(t) \\ t^2 & 2rt \end{bmatrix} \quad (3.2.2)$$

ヤコビアンの計算処理が定義できたので、単変数の場合同様に反復で用いられる式を生成するための makeMultiIteration 関数を定義してみます。

```
> makeMultiIteration := proc(flist, vlist)
    # flist: 式のリスト, vlist: 変数のリスト
    uses LinearAlgebra;
    local Jinv, fret;

    # ヤコビアンとその逆行列を計算
    Jinv := MatrixInverse(jacob(flist,vlist));
    # 反復で使われる式を算出
    fret := vlist - convert(Jinv.Vector(flist), list);
    unapply(fret, vlist);
  end proc:
```

さっそく次の円と曲線の交点を求める問題に適用してみます。

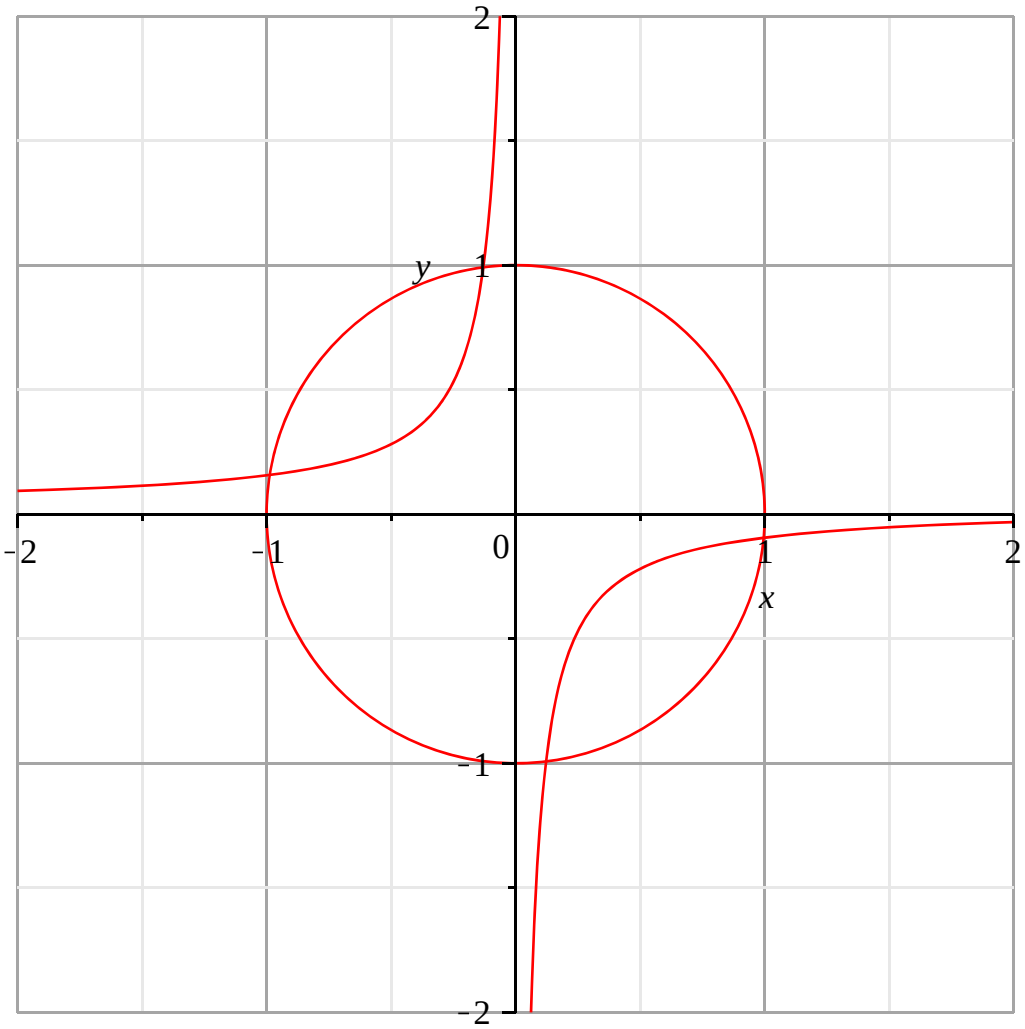
```
> F := [x^2+y^2-1,3.2*x*y-0.1*x+0.4];
```

(3.1)

$$F := [x^2 + y^2 - 1,\ 3.2\,x\,y - 0.1\,x + 0.4] \qquad \textbf{(3.1)}$$

この F を図示すると次のようになります。

```
> plots[implicitplot](F,x=-2..2,y=-2..2,
    scaling=constrained,gridlines,gridrefine=3
  );
```



makeMultiIteration 関数を用いて反復のための式を生成します。

```
> newton_sys := makeMultiIteration(F,[x,y]);
```

$$newton\_sys := (x, y) \mapsto \left[-\frac{3.2\,x\,(x^2 + y^2 - 1)}{6.4\,x^2 - 6.4\,y^2 + 0.2\,y} + \frac{2\,y\,(3.2\,x\,y - 0.1\,x + 0.4)}{6.4\,x^2 - 6.4\,y^2 + 0.2\,y} + x,\right. \qquad \textbf{(3.2)}$$

$$\frac{(3.200000000\,y - 0.1000000000)\,(x^2 + y^2 - 1)}{6.400000000\,x^2 - 6.400000000\,y^2 + 0.2000000000\,y} - \frac{2\,x\,(3.2\,x\,y - 0.1\,x + 0.4)}{6.4\,x^2 - 6.4\,y^2 + 0.2\,y}$$

$$\left. + y \right]$$

$[x, y]$ に対して適当な初期値から反復計算を行ってみます；

```
> x0list := [-0.5,0.7];
  to 10 do
    x0list := newton_sys(x0list[]);
  end do;
```

$$x0list := [-0.5,\ 0.7]$$

$$x0list := [0.4699140401,\ 1.578510028]$$

$$x0list := [0.0455930989,\ 1.162382048]$$

$$x0list := [-0.1046221728, 1.016340048]$$
$$x0list := [-0.1294568299, 0.9921899598]$$
$$x0list := [-0.1301750593, 0.9914915319]$$
$$x0list := [-0.1301756610, 0.9914909466]$$
$$x0list := [-0.1301756611, 0.9914909466]$$
$$x0list := [-0.1301756611, 0.9914909467]$$
$$x0list := [-0.1301756610, 0.9914909467]$$
$$x0list := [-0.1301756610, 0.9914909467] \quad (3.3)$$

得られた値は、Maple の fsolve コマンドによる結果とも非常に近い結果が得られています。

```
> fsolve(F,{x=-0.5,y=0.7});
```

$$\{x = -0.1301756611, y = 0.9914909466\} \quad (3.4)$$

makeMultiIteration 関数を用いて、ニュートン法を行うプログラム例を以下に記述しておきます。

```
> MultiNewtonIteration := proc(flist, vlist, inilist, ite)
    local nf, x0, tmp;

    nf := makeMultiIteration(flist,vlist);
    x0 := inilist;
    to ite do
      tmp := x0;
      x0 := nf(x0[]);
      if max(map(abs,x0-tmp))<10.0^(-Digits) then break; end
  if;
    end do;
  end proc;
```

MultiNewtonIteration := **proc**(*flist*, *vlist*, *inilist*, *ite*) (3.5)

local *nf*, *x0*, *tmp*;

nf := *makeMultiIteration*(*flist*, *vlist*);

x0 := *inilist*;

to *ite* **do**

tmp := *x0*;

x0 := *nf* (*x0*[ ]);

if max(*map*(abs, *x0* − *tmp*)) < 10.0^(− *Digits*) **then break end if**

end do

end proc

F の異なる零点を求めてみます。別の初期値を指定してニュートン法の反復を行ってみます。

```
> MultiNewtonIteration(F,[x,y],[-5.4,2.8],100);
```

$$[-0.9874652220, 0.1578367367] \quad (3.6)$$

```
> MultiNewtonIteration(F,[x,y],[10.4,1.8],100);
```

$$[0.9955429141, -0.09430963005] \quad (3.7)$$

unapply コマンドを用いることで、ニュートン法をより簡潔に記述することが出来ると共に、反復に用いる数式も明示的に算出することができるため、数値演算の誤差解析にも適用することが可能です。
なお、多変数多項式の零点の計算については、グレブナ基底を用いた厳密な方法も適用することが可能です。こちらの資料を参考にしてください。