Automatic differentiation with Perl

by J-L Morel - (jl_morel@bribes.org)

http://www.bribes.org/perl/

1 Introduction

For some numerical algorithms, it is necessary to calculate the derivative of a function at a particular point.
For instance, if the function f is continuously differentiable, the numerical sequence defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

converges to a solution of the equation $f(x) = 0$ for a good initial value x_0.

The problem is how to compute $f'(x_n)$ when we only know $f(x)$?

We can calculate the derivative symbolically, by hand or with a computer algebra system. It is then necessary to provide $f(x)$ and $f'(x)$ to the program.

Another solution is to calculate an approximation of $f'(x_n)$. The simplest formula is

$$f'(x_n) \approx \frac{f(x_n + h) - f(x_n)}{h}$$

for some appropriately small h. These calculations can be made by the program: it is only necessary to provide $f(x)$ to it. But the precision is not always guaranteed.

With *Automatic Differentiation* (AD) it is possible to calculate at the same time $f(x_n)$ and $f'(x_n)$ step by step with the maximum accuracy.

2 Principle

If we know the *numerical values* $u(x_0)$ and $v(x_0)$ of two functions and their derivatives $u'(x_0)$ and $v'(x_0)$ at a point x_0, then one can calculate

- $(u+v)(x_0) = u(x_0) + v(x_0)$ and $(u+v)'(x_0) = u'(x_0) + v'(x_0)$
- $(u-v)(x_0) = u(x_0) - v(x_0)$ and $(u+v)'(x_0) = u'(x_0) - v'(x_0)$
- $(u*v)(x_0) = u(x_0) * v(x_0)$ and $(u*v)'(x_0) = u'(x_0) * v(x_0) + u(x_0) * v'(x_0)$
- $(u\ /\ v)(x_0) = u(x_0)\ /\ v(x_0)$ and $(u\ /\ v)'(x_0) = \big(u'(x_0) * v(x_0) - u(x_0) * v'(x_0)\big)\ /\ v^2(x_0)$

It's commonplace!

If we consider the pairs of real numbers $(u(x_0), u'(x_0))$ and $(v(x_0), v'(x_0))$, we can rewrite the preceding rules symbolically

- $\big(u(x_0)\ ,\ u'(x_0)\big) + \big(v(x_0)\ ,\ v'(x_0)\big) = \big(u(x_0) + v(x_0)\ ,\ u'(x_0) + v'(x_0)\big)$
- $\big(u(x_0)\ ,\ u'(x_0)\big) - \big(v(x_0)\ ,\ v'(x_0)\big) = \big(u(x_0) - v(x_0)\ ,\ u'(x_0) - v'(x_0)\big)$
- $\big(u(x_0)\ ,\ u'(x_0)\big) * \big(v(x_0)\ ,\ v'(x_0)\big) = \big(u(x_0) * v(x_0)\ ,\ u'(x_0) * v(x_0) + u(x_0) * v'(x_0)\big)$
- $\big(u(x_0)\ ,\ u'(x_0)\big)\ /\ \big(v(x_0)\ ,\ v'(x_0)\big) = \big(u(x_0)\ /\ v(x_0)\ ,\ \big(u'(x_0) * v(x_0) - u(x_0) * v'(x_0)\big)\ /\ v^2(x_0)\big)$

Moreover, if f is an elementary function (sin, cos..), we have for the composition of functions $(f \circ u)(x_0) = f(u(x_0))$ and $(f \circ u)'(x_0) = f'(u(x_0)u'(x_0)$. We can write symbolically

$$f\big((u(x_0)\ ,\ u'(x_0)\big) = \big(f(u(x_0))\ ,\ f'(u(x_0))u'(x_0)\big)$$

The question is now: how to implement these operations in Perl?

3 Implementation

We can code a pair of real numbers by an array and then overload the operators to do the preceding operations on the pairs of numbers.

It is convenient to put all these functions in a module. We name it `AutoDiff`.

⟨*AutoDiff.pm* 2a⟩ ≡ ⟨2a⟩ ▽3a

```
use strict;
use warnings;
package AutoDiff;

use Carp;

sub new{                            # the constructor
  my $class = $_[0];
  my $var;
  if (@_ == 3) {
    $var = [ $_[1], $_[2] ];            # 2 arguments [u, u']
  }
  elsif (@_ == 2) {
    $var = [ $_[1], 1 ];                # 1 argument [u, 1]: var
  }
  else {
    $var = [ 0, 0 ];                    # no argument [0,0]: var
  }
  bless $var, $class;
}

sub get_val {                       # accessor to the value
  $_[0]->[0];
}

sub set_val {
  $_[0]->[0] = $_[1];
}

sub get_der {                       # accessor to the derivative
  $_[0]->[1];
}

sub set_der {
  $_[0]->[1] = $_[1];
}
```

The constructor can take 0,1 or 2 arguments. For instance:

- `$x = AutoDiff->new();`
  $x is a variable with value 0.
- `$x = AutoDiff->new(5);`
  $x is a variable with value 5.
- `$x = AutoDiff->new(5,0);`
  $x is a constante with value 5 (because is derivative is 0).

We define two accessors to read the values of $f(x)$ and of $f'(x)$ at the end of calculation.
A small test:

⟨*test01.pl* 2b⟩ ≡ ⟨2b⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use AutoDiff;
```

```
my $x = AutoDiff->new();
print "( ", $x->get_val, " , ", $x->get_der, " )\n";

$x = AutoDiff->new(3);
print "( ", $x->get_val, " , ", $x->get_der, " )\n";

$x = AutoDiff->new(3,5);
print "( ", $x->get_val, " , ", $x->get_der, " )\n";
```

Output:

```
( 0 , 0 )
( 3 , 1 )
( 3 , 5 )
```

Now that we have our objects, we write the methods.

The method `as_string` will be called to print an object in the format `(a , b)`. The four other methods implements the arithmetic operations. If the second argument is not a reference, it's a number and it is converted: a becomes $(a,0)$. For the noncommutative operations (− and /), if the arguments are permuted one puts them in the right order.

⟨*AutoDiff.pm* 3a⟩ ≡

⟨3a⟩ △2a ▽4a

```
sub as_string{
  my $self = shift;
  return "($self->[0] , $self->[1])";
}

sub _add {                                  # addition
  my ($u, $v) = @_;
  $v = AutoDiff->new($v, 0) unless ref $v;
  AutoDiff->new(
    $u->[0] + $v->[0],                        # u + v
    $u->[1] + $v->[1]                         # u' + v'
  );
}

sub _sub {                                  # substraction
  my ($u, $v, $mut) = @_;
  $v = AutoDiff->new($v, 0) unless ref $v;
  if ($mut) {
    $mut=$u; $u=$v; $v=$mut;                  # swap $u and $v ($mut temp. var)
  }
  AutoDiff->new(
    $u->[0] - $v->[0],                        # u - v
    $u->[1] - $v->[1]                         # u' - v'
  );
}

sub _mul {                                  # multiplication
  my ($u, $v) = @_;
  $v = AutoDiff->new($v, 0) unless ref $v;
  AutoDiff->new(
    $u->[0] * $v->[0],                        # uv
    $u->[1] * $v->[0] + $u->[0] * $v->[1]     # u'v + uv'
  );
}

sub _div {                                  # division
  my ($u, $v, $mut) = @_;
  $v = AutoDiff->new($v, 0) unless ref $v;
```

```
  if ($mut) {
    $mut=$u; $u=$v; $v=$mut;                          # swap $u and $v ($mut temp. var)
  }
  AutoDiff->new(
    $u->[0] / $v->[0],                                # u/v
    ($u->[1] - $u->[0]*$v->[1]/$v->[0])/$v->[0]       # (u' - uv'/v)/v
  );
}
```

There are two cases for the operator **:

- if $u > 0$ then $u^v = e^{v\log(u)}$ and we have

$$\left(u^v\right)' = \left(e^{v\log(u)}\right)' = \left(e^{v\log(u)}\right)\left(v\log(u)\right)' = u^v\left(v'\log(u) + v\frac{u'}{u}\right)$$

- if $u \le 0$ then v must be an integer and $(u^v)' = vu^{v-1}$.
  If v is not an integer or if $v' \neq 0$ then the calculation of `u**v` generates an error message and the program terminates.

⟨*AutoDiff.pm* 4a⟩ ≡

⟨4a⟩ △3a ▽4b

```
sub _pow {                                    # power **
  my ($u, $v, $mut) = @_;
  my $r = AutoDiff->new();
  $v = AutoDiff->new($v, 0) unless ref $v;
  if ($mut) {
    $mut=$u; $u=$v; $v=$mut;                    # swap $u and $v ($mut temp. var)
  }
  $r->[0] =  $u->[0]**$v->[0];
  if ($u->[0] > 0) {
    $r->[1] =  ($v->[1]*log($u->[0])+$v->[0]/$u->[0]*$u->[1])*$r->[0];
  }
  else {
    croak "Illegal power exponent" if $v->[1]!=0 or $v->[0]!=int($v->[0]);
    $r->[1] = $v->[0]*$u->[0]**($v->[0]-1)*$u->[1];
  }
  $r;
}
```

Now we redefine the numerical functions of Perl which we want to overload.

⟨*AutoDiff.pm* 4b⟩ ≡

⟨4b⟩ △4a ▽5a

```
sub _sin {
  my $x = shift;
  AutoDiff->new(
    sin($x->[0]),                             # sin(u)
    cos($x->[0])*$x->[1]                      # cos(u)u'
  );
}

sub _cos {
  my $x = shift;
  AutoDiff->new(
    cos($x->[0]),                             # cos(u)
    -sin($x->[0])*$x->[1]                     # -sin(u)u'
  );
}

sub _sqrt {
  my $x = shift;
  AutoDiff->new(
```

```
    sqrt($x->[0]),                          # √u
    $x->[1]/2/sqrt($x->[0])                 # u'/(2√u)
  );
}

sub _exp {
  my $x = shift;
  AutoDiff->new(
    exp($x->[0]),                           # exp u
    exp($x->[0])*$x->[1]                    # exp u u'
  );
}

sub _log {
  my $x = shift;
  AutoDiff->new(
    log($x->[0]),                           # log(u)
    $x->[1]/$x->[0]                         # u'/u
  );
}
```

Finally, let's overload!

⟨*AutoDiff.pm* 5a⟩ ≡ ⟨5a⟩ △4b ▽5b

```
use overload
  '"'"   => 'as_string',
  '+'    => '_add',
  '-'    => '_sub',
  '*'    => '_mul',
  '/'    => '_div',
  '**'   => '_pow',
  'sin'  => '_sin',
  'cos'  => '_cos',
  'sqrt' => '_sqrt',
  'exp'  => '_exp',
  'log'  => '_log';
```

Of course, we could add other functions, but it's enought for a demonstration module.

A Perl module must end with a true value or else it is considered not to have loaded.

⟨*AutoDiff.pm* 5b⟩ ≡ ⟨5b⟩ △5a

```
1;
```

Let f be the function defined by

$$f(x) = x\cos x + \frac{e^{3x-1}}{\sqrt{\log(x^2+1)}}$$

We have

$$f'(x) = \cos x - x\sin x + \frac{\left(3\ln\left(x^2+1\right) - x + 3x^2\ln\left(x^2+1\right)\right)e^{3x-1}}{\left(\ln^{\frac{3}{2}}\left(x^2+1\right)\right)\left(x^2+1\right)}$$

and, for example:

$$f\left(\frac{3}{2}\right) = \frac{3}{2}\cos\frac{3}{2} + \frac{e^{\frac{7}{2}}}{\sqrt{\ln\frac{13}{4}}} \approx 30.609$$

$$f'\left(\frac{3}{2}\right) = \cos\frac{3}{2} - \frac{3}{2}\sin\frac{3}{2} + \frac{4}{13}\frac{e^{\frac{7}{2}}}{\ln^{\frac{3}{2}}\frac{13}{4}}\left(\frac{39}{4}\ln\frac{13}{4} - \frac{3}{2}\right) \approx 78.138$$

Let's write a program to find these values.

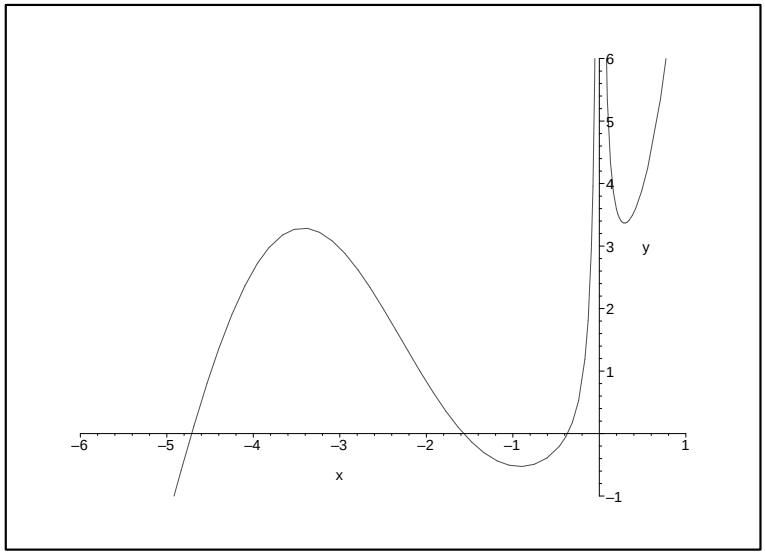
⟨*test02.pl* 6a⟩ ≡ (6a)

```
#!/usr/bin/perl
use strict;
use warnings;
use AutoDiff;

sub f{                                  # Fortunately, we code only f(x)!
  my $x=shift;
  $x*cos($x)+exp(3*$x-1)/sqrt(log($x**2+1));
}

my $x = AutoDiff->new(1.5);
my $y = f($x);
print "f(1.5)  = ", $y->get_val, "\n";
print "f'(1.5) = ", $y->get_der, "\n";
```

Output:

```
f(1.5)  = 30.6087392558198
f'(1.5) = 78.1381547733882
```

It's work! We obtain the numerical value $f'(1.5)$ of the derivative without using the expression of $f'(x)$.
Here is the graph of the function f:



Let's use the Newton's method to find the root of $f(x) = 0$ which is close to -0.5

⟨*test03.pl* 6b⟩ ≡ (6b)

```
#!/usr/bin/perl
use strict;
use warnings;
use AutoDiff;

sub f{
  my $x=shift;
  $x*cos($x)+exp(3*$x-1)/sqrt(log($x**2+1));
}

my $x0 = -0.5;

for( 1..10) {
  my $x = AutoDiff->new($x0,1);
  my $y = f($x);
  print "f($x0) = ", $y->get_val, "\n";
```

```
  $x0 -= $y->get_val/$y->get_der;                    # x_{n+1} = x_n - f(x_n)/f'(x_n)
}
```

Output:

```
f(-0.5) = -0.265022551785238
f(-0.319794469958085) = 0.148105761721095
f(-0.361518303676643) = 0.0166893910729097
f(-0.367488074604522) = 0.000263528115675593
f(-0.36758538684753) = 6.77893827183418e-008
f(-0.367585411892797) = 4.55191440096314e-015
f(-0.367585411892799) = 1.11022302462516e-016
f(-0.367585411892799) = 1.11022302462516e-016
f(-0.367585411892799) = 5.55111512312578e-017
f(-0.367585411892799) = 5.55111512312578e-017
```

The sought root is $x^* \approx -0.3675854118928$.

4 Dual numbers

The algorithm implemented by our module uses only numbers – it does not make any symbolic calculus.
It does operations on couples of real numbers according to the definitions

$$(a,a') + (b,b') = (a+a', b+b') \qquad (a,a') - (b,b') = (a-a', b-b')$$
$$(a,a') * (b,b') = (aa', ab' + a'b) \qquad (a,a') \,/\, (b,b') = \left(\frac{a}{b}, \frac{a'b-ab'}{b^2}\right)$$

(a, a', b and b' are real numbers and $b \neq 0$ for the last definition).
In this system, one identifies a real number x with the couple $(x,0)$. One can then write

$$(x,y) = (x,0) + (0,y) = x(1,0) + y(0,1) = x + y\varepsilon \qquad \text{where} \qquad \varepsilon = (0,1)$$

Let us notice that $\varepsilon^2 = (0,1) * (0,1) = (0\times 0, 0\times 1 + 1\times 0) = (0,0) = 0$

This kind of complex numbers $x + y\varepsilon$ with $\varepsilon^2 = 0$ are called *dual numbers* (A. Neumaier in [2] call them *differential numbers*).

The set of dual numbers with the operations defined above forms a commutative and associative ring.

With our module `AutoDiff` we can do calculations with the dual numbers. Let us say to solve the quadratic equation $(2+5\varepsilon)x^2 - (6+13\varepsilon)x + (4+12\varepsilon) = 0$

⟨*test04.pl* 7a⟩ ≡

```
#!/usr/bin/perl                                                    ⟨7a⟩
use strict;
use warnings;
use AutoDiff;

my $e = AutoDiff->new(0,1);                        # $e = ε

my $a = 2+5*$e;                                    # a = 2 + 5ε
my $b = 6-13*$e;                                   # b = 6 + 13ε
my $c = 4+12*$e;                                   # c = 4 + 12ε

my $d = $b**2-4*$a*$c;                             # Δ = b^2 - 4ac
print "Discriminant = $d\n";

my $x1 = (-$b-sqrt($d))/(2*$a);                    # x_1 = (-b-√Δ)/2a
my $x2 = (-$b+sqrt($d))/(2*$a);                    # x_2 = (-b+√Δ)/2a

print "x1 = $x1 and x2 = $x2\n";
```

Output:

```
Discriminant = (4 , -332)
x1 = (-2 , 29) and x2 = (-1 , -15)
```

One thus has $x_1 = -2 + 29\varepsilon$ and $x_2 = -1 - 15\varepsilon$.

For more on the dual numbers, see the Wikipedia page http://en.wikipedia.org/wiki/Dual_number
The book of I.M. Yaglom [3] (a classic) gives a geometrical interpretation of dual numbers as oriented lines of a plane.

5 Jacobian

We can also calculate partial derivatives of a function of several variables with the module `AutoDiff`.
Let f be the function defined by $f(x,y) = x^3y^2 + xy$. We can evaluate the partial derivatives of f at the point $(1,2)$.

$$\frac{\partial f}{\partial x}(x,y) = 3x^2y^2 + y \qquad \frac{\partial f}{\partial x}(1,2) = 14$$

$$\frac{\partial f}{\partial y}(x,y) = 2x^3y + x \qquad \frac{\partial f}{\partial y}(1,2) = 5$$

We can check with this program:

⟨*test05.pl* 8a⟩ ≡ ⟨8a⟩

```
#!/usr/bin/perl
use strict;
use warnings;
use AutoDiff;

sub f {
  my ($x, $y) = @_;
  $x**3 * $y**2 + $x * $y;
  }

my $x = AutoDiff->new(1,1);                    # because ∂x/∂x = 1
my $y = AutoDiff->new(2,0);                    # because ∂y/∂x = 0
print "f'x = ", f($x,$y)->get_der, "\n";

$x->set_der(0);                                # because ∂y/∂x = 0
$y->set_der(1);                                # because ∂y/∂y = 1
print "f'y = ", f($x,$y)->get_der, "\n";
```

Output:

```
f'x = 14
f'y = 5
```

If a function f from $\mathbb{R}^n$ to $\mathbb{R}^m$ is given by its m component functions f_1, f_2, ... f_m then

$$f(x_1, x_2, \ldots x_n) = \begin{pmatrix} f_1(x_1, x_2, \ldots x_n) \\ f_2(x_1, x_2, \ldots x_n) \\ \cdots \\ f_m(x_1, x_2, \ldots x_n) \end{pmatrix} \in \mathbb{R}^m$$

and its *Jacobian matrix* is

$$\mathrm{J}(f; x_1, x_2, \ldots x_n) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1, x_2, \ldots x_n) & \frac{\partial f_1}{\partial x_2}(x_1, x_2, \ldots x_n) & \cdots & \frac{\partial f_1}{\partial x_n}(x_1, x_2, \ldots x_n) \\ \frac{\partial f_2}{\partial x_1}(x_1, x_2, \ldots x_n) & \frac{\partial f_2}{\partial x_2}(x_1, x_2, \ldots x_n) & & \frac{\partial f_2}{\partial x_n}(x_1, x_2, \ldots x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x_1, x_2, \ldots x_n) & \frac{\partial f_m}{\partial x_2}(x_1, x_2, \ldots x_n) & \cdots & \frac{\partial f_m}{\partial x_n}(x_1, x_2, \ldots x_n) \end{pmatrix}$$

Let's write a routine to calculate a Jacobian matrix.
The routine receives in input a reference to the function f and an array of the numerical values of the variables. It calculates the Jacobian matrix row by row and pushes them in an array `@j`. This array `@j` is converted into a matrix: we use the module `Math::Matrix`.

⟨*Calculate the Jacobian matrix* 8b⟩ ≡ ⟨8b⟩

```
sub Jacobian {
  my $rf = shift;                              # reference to the function f
```

```
  my $v = shift;                                  # reference to the array of variables
  my $dv = [ map {AutoDiff->new($_, 0)} @{$v}];   # all variables as constantes here!
  my @j;                                          # array to receive the rows of J
  foreach my $i (0..$#$v) {                      # for all the n variables, one by one
    $dv->[$i]->set_der(1);                        # partial derivative with respect to x_i
    push @j, [map {$_->[1]} @{$rf->($dv)}];       # calculate column i and save it in @j
    $dv->[$i]->set_der(0);                        # reset
  }
  Math::Matrix->new(@j)->transpose;               # transform @j into matrix and transpose it
}
```

Examples:

1. $f(x,y,z) = x^2+2y^2-z^2 \quad \mathrm{J}(f;x,y,z) = \big(2x,4y,-2z\big) \quad \mathrm{J}(f;1,1,1) = (2,4,-2)$

2. $g(x) = \begin{pmatrix} x \\ x^2 \\ x^3 \end{pmatrix} \quad \mathrm{J}(g;x) = \begin{pmatrix} 1 \\ 2x \\ 3x^2 \end{pmatrix} \quad \mathrm{J}(g;1) = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$

3. $h(x,y,z) = \begin{pmatrix} x^2+y^2+z^2-1 \\ 2x^2+y^2-4z \\ 3x^2-4y+z^2 \end{pmatrix} \quad \mathrm{J}(h;x,y,z) = \begin{pmatrix} 2x & 2y & 2z \\ 4x & 2y & -4 \\ 6x & -4 & 2z \end{pmatrix} \quad \mathrm{J}(h;1,1,1) = \begin{pmatrix} 2 & 2 & 2 \\ 4 & 2 & -4 \\ 6 & -4 & 2 \end{pmatrix},$

We can now test our routine.

⟨*test06.pl* 9a⟩ ≡

```
#!/usr/bin/perl                                                    ⟨9a⟩
use strict;
use warnings;
use AutoDiff;
use Math::Matrix;
```

⟨*Calculate the Jacobian matrix* 8b⟩

```
sub f {                                 # function R^3 -> R
  my ($x, $y, $z) = @{$_[0]};
  return [$x**2+2*$y**2-$z**2];
}
my $v = [1, 1, 1];

print "Example 1:\n";
print Jacobian(\&f, $v),"\n";


sub g {                                 # function R -> R^3
  my ($x) = @{$_[0]};
  return [$x,
          $x**2,
          $x**3];
}
$v = [1];

print "Example 2:\n";
print Jacobian(\&g, $v),"\n";

sub h {                                 # function R^3 -> R^3
  my ($x, $y, $z) = @{$_[0]};
  return [ $x**2+$y**2+$z**2-1,
           2*$x**2+$y**2-4*$z,
           3*$x**2-4*$y+$z**2
         ];
}
$v = [1, 1, 1];
```

```
print "Example 3:\n";
print Jacobian(\&h, $v),"\n";
```

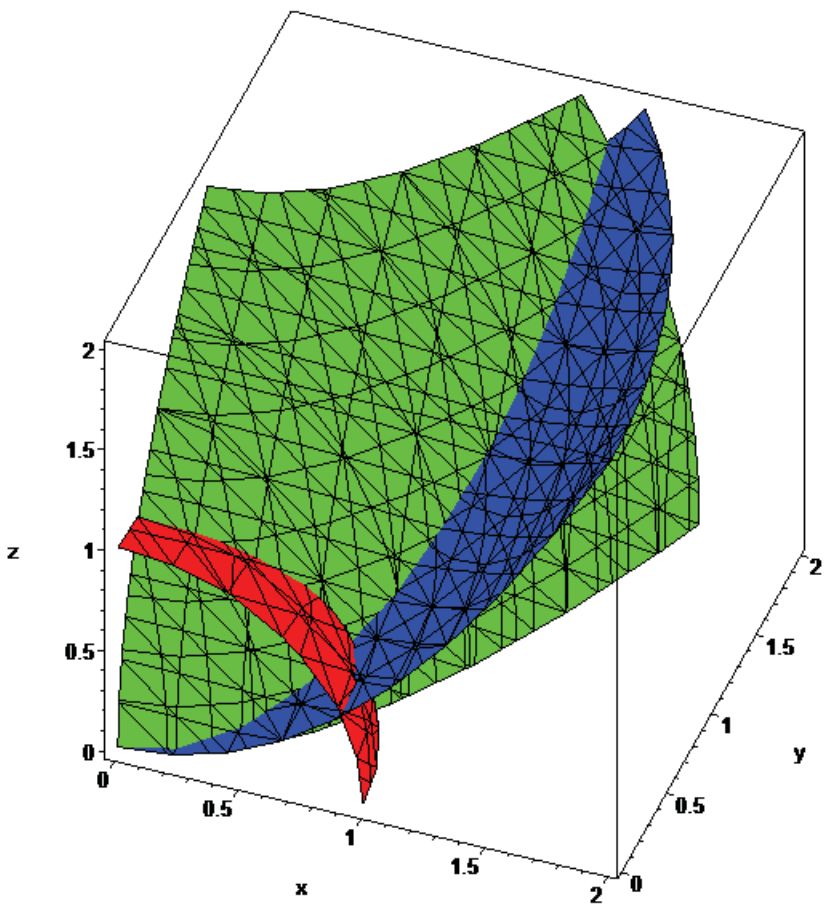
Output:

```
Example 1:
   2.00000    4.00000   -2.00000

Example 2:
   1.00000
   2.00000
   3.00000

Example 3:
   2.00000    2.00000    2.00000
   4.00000    2.00000   -4.00000
   6.00000   -4.00000    2.00000
```

Let us suppose now that we want to solve the system of nonlinear equations

$$\begin{cases} x^2+y^2+z^2-1=0 \\ 2x^2+y^2-4z=0 \\ 3x^2-4y+z^2=0 \end{cases}$$

by using the Newton's method.
The graphic representation of the system functions shows a solution close to the point $(0.5, 0.5, 0.5)$.



Here, f is the function from $\mathbb{R}^3$ to $\mathbb{R}^3$ defined by

$$f(x,y,z) = \left(x^2+y^2+z^2-1,\ 2x^2+y^2-4z,\ 3x^2-4y+z^2\right)$$

The Newton's sequence associated is defined by

$$v_{n+1}^t = v_n^t - \mathrm{J}^{-1}(f;v_n)f(v_n)^t$$

where $v_n = (x_n, y_n, z_n)$ and $v_0 = (0.5, 0.5, 0.5)$.
Because v_n and $f(v_n)$ are written as one-forms (row matrix), it is necessary to transform them into vectors (column matrix) by transposition.

⟨*test07.pl* 11a⟩ ≡

```
#!/usr/bin/perl
use strict;
use warnings;
use AutoDiff;
use Math::Matrix;

⟨Calculate the Jacobian matrix 8b⟩

sub f {                                 # function R^3 -> R^3
  my ($x, $y, $z) = @{$_[0]};
  return [ $x**2+$y**2+$z**2-1,
           2*$x**2+$y**2-4*$z,
           3*$x**2-4*$y+$z**2
         ];
}

my $v = [0.5, 0.5, 0.5];
print "x = $v->[0] y = $v->[1] z = $v->[2]\n";

for (1..6) {
  my $x = Math::Matrix->new($v)->transpose;
  my $fv = Math::Matrix->new(f($v))->transpose;
  my $j = Jacobian(\&f, $v);
  $x = $x - $j->invert*$fv;
  $v = [@{$x->transpose->[0]}];
  print "x = $v->[0] y = $v->[1] z = $v->[2]\n";
}
```

⟨11a⟩

Output:

```
x = 0.5 y = 0.5 z = 0.5
x = 0.875 y = 0.5 z = 0.375
x = 0.789816602316602 y = 0.496621621621622 z = 0.369932432432432
x = 0.785210443444361 y = 0.496611393007268 z = 0.369922830787265
x = 0.785196933178586 y = 0.496611392944656 z = 0.369922830745872
x = 0.785196933062355 y = 0.496611392944656 z = 0.369922830745872
x = 0.785196933062355 y = 0.496611392944656 z = 0.369922830745872
```

The solutions of our system are roughly $x \approx 0.785196933062355$, $y \approx 0.496611392944656$, $z \approx 0.369922830745872$.

6 Conclusion

In this paper we proposed only a Perl implementation of the *forward mode* automatic differentiation. Another possibility is *reverse mode* AD, much more difficult to implement. To be continued...

A good starting point about AD is the Wikipedia page:
http://en.wikipedia.org/wiki/Automatic_differentiation
with a lot of links.
See also the Community Portal for Automatic Differentiation:
http://www.autodiff.org/

References

[1] Andreas Griewank and Andrea Walther. *Evaluating Derivatives - Principles and Techniques of Algorithmic Differentiation.* SIAM, second edition, 2008.

[2] Arnold Neumaier. *Introduction to Numerical Analysis.* Cambridge University Press, 2001.

[3] I. M. Yaglom. *Complex Numbers in Geometry.* Academic Press, 1968.