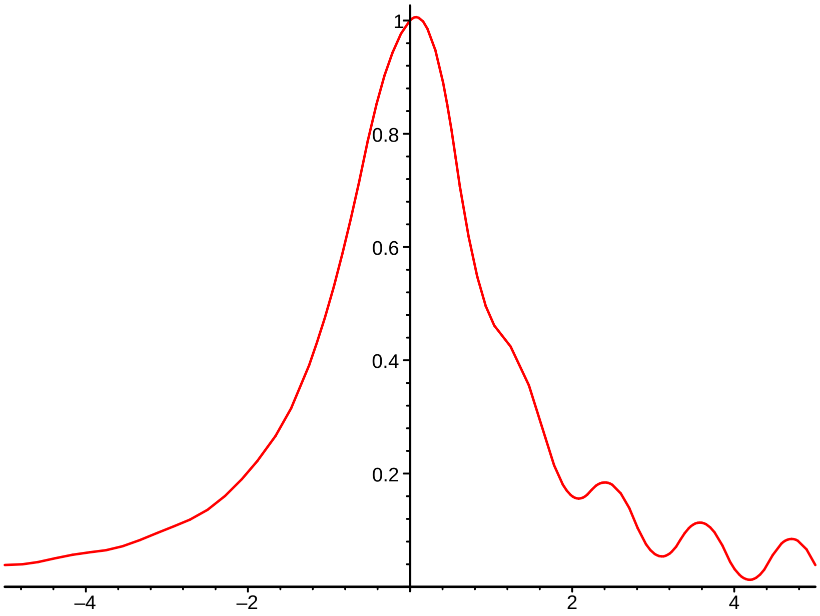
Exporting Maple Words to C++

Ma 777 Spring 1998



April 24, 1998

Contents

1 First Steps

1.1 Ways to translate Chebyshef

Often, we will solve a problem in Maple, then try to translate the solution into C++.

1.1.1 ***Problem:*** *There are at least 4 translations of the first syllable of Chebyshef: Cheb, Ceb, Tscheb, and Tcheb. Also the y can be replaced with an i, and the last syllable has 5 translations: shef, schef, cev, cheff, and scheff. Write a program to list all resulting spellings of the man's name.*

Here is a solution in Maple.

```
front := [`Cheb`,`Ceb`,`Tscheb`,`Tcheb`];
middle := [`y`,`i`];
back := [`shef`,`schef`,`cev`,`cheff`,`scheff`];
variations := NULL:
for f in front do for m in middle do for
b in back do variations := variations,cat(f,m,b) od od od:
variations;
```

1.1.2 ***Exercise:*** *Work this problem in some other language, say C++.*

Solution: Here is something that works in C++. *If you are on a t machine*, copy this to a text file opened with vi and save it to say cheby.cc. Then compile it with the gnu compiler by entering

$$g{+}{+}\ cheby.cc$$

You should get no error messages, and the compiler leaves a file called a.out in the directory you are in. To execute the program, enter a.out

```
#include <iostream.h>
//
char* front[]={"Cheb","Ceb","Tscheb","Tcheb"};
char* middle[]={"y","i"};
char* back[]={"shef","schef","cev","cheff","scheff"};
//
main()
{
for (int i=0;i<4;i++)
for (int j=0;j<2;j++)
for (int k=0;k<5;k++)
cout << front[i]
<< middle[j] <<
back[k] << ", ";
return(0);
}
```

1.1.3 ***Modify*** *the Maple solution so the spellings are enumerated, that is, instead of the output Chebyshef, Chebyschef, Chebycev, ..., we have the output 1 Chebyshef, 2 Chebyschef, 3 Chebycev, ...*

1.1.4 ***Modify*** *the C++ solution so that it is an ansi C solution. (ie replace iostream.h and cout by the C version.)*

1.2 Numerical Differentiation

1.2.1 ***Problem.*** *Suppose you estimate the derivative of a function* f *at a point* a *by the sequence* $a_n = \frac{\mathrm{f}(a+h^n)-\mathrm{f}(a)}{h^n}$*, where* h *is chosen between 0 and 1. The calculations are made with a fixed precision, say 10 digits. So, in theory, as n gets large,* a_n *gets close to* $\mathrm{f}\,'(a)$*. But something goes wrong. What is it?*

Initial Analysis: We can predict what will go wrong. Depending on the function, the difference will become 0 when the two terms agree to the first 10 digits, so the estimate will flip to 0.

Test this in Maple using the function sin at a=0.2, with h = .01.

```
f := sin;
a := 0.2;
h := .01;
for n from 1 to 10 do (f(a+h^n)-f(a))/h^n od;
```

As we predicted, the answer flips to 0 (before 6 digits of accuracy is obtained; the calculations are done with 10 digits by default).

```
cos(.2);
```

It is easy to define a procedure in Maple. Just copy the input cells you have used into one cell, put in a proc line and append an end at the bottom and voila, a word is born. For example,

```
ndiv := proc(f,a,inc,n)
local h,i;
h := evalf(inc);
for i from 1 to n do
print( a+h^i, evalf((f(a+h^i)-f(a))/h^i));
od;
NULL:
end:
```

```
ndiv(sin,0.2,.01,10);
```

```
evalf(cos(.2));
```

Analysis: The best estimate of the derivative in the sequence is when h is $10^{(-5)}$. After that, loss of significant digits occur in the numerator due to 'catastropic subtraction'.

Converting Maple procs to Ansi C: You can convert simple procedures written in Maple to Ansi C code using the C-word in Maple. This is not part of the standard vocabulary, but is loaded in with the package codegen.

```
with(codegen);
```

First, let's rewrite ndiv to make the local variable h replace the input variable inc.

```
ndiv := proc(f,a,h,n)
local i;
```

```
for i from 1 to n do
print( a+h^i, evalf((f(a+h^i)-f(a))/h^i));
od;
NULL:
end:
```

Now convert ndiv.

```
C(ndiv,ansi);
```

As it stands, it is just a procedure poorly defined in ansi C. Poorly defined, because for one thing ansi C doesn't know about 'evalf'. Also, in the top line the reference to 'double f' should read 'double f(double)' since f is a function which takes a real argument.

Now, we need a program that uses ndiv. Here is a very simple one.

```
// Problem.   Use ndiv in a C++ program to estimate the derivative of
sin(x) + exp(x*x)
//      at x   =.2 .
//  Solution
//   the include directives.  Nearly always you will need to include
iostream.h.  We are including //   math.h because we use the sin and
exp functions in our test.
//
#include <iostream.h>
#include <math.h>
//
//   The modified version of ndiv that we exported from Maple.
double ndiv(double f(double),double a,double h, int n)
{
int i;
 for(i = 1;(i <= n);i++)
    printf( ”%e  %e\n”
,a+pow(h,i),((f(a+pow(h,i))-f(a))/pow(h,i)));
  return(NULL);
}
//
//  Here is the test function
double g(double x)
{
return(sin(x)+exp(x*x));
}
//
//  Here is the main().
main()
{
cout   <<   ndiv(g, .2, .01, 12) ;
return(0);
}
```

Use the vi editor to copy this to a file ndiff.cpp. Then, in a t-machine execute the command *g++ ndiff.cpp*

You should get no compiler errors. Now just type a.out and press return to run the program. Note that the flip to 0 phenomenon occurs as it should.

1.2.2 *Problem. Modify ndiff.cpp to also compute and print out the estimate* $\frac{\mathrm{f}(a+h)-\mathrm{f}(a-h)}{2\,h}$ *for comparison with the standard estimate* $\frac{\mathrm{f}(a+h)-\mathrm{f}(a)}{h}$.

2 Approximating functions with polynomials

2.1 A typical problem

2.1.1 ***Problem:*** *Given a function $y = f(x)$ which is analytic on the interval [a,b], use Taylor's theorem with remainder to find a polynomial $y = \mathrm{p}(x)$ that approximates f to within a specified error on the interval. Then write a procedure to implement the polynomial function you have found. Then export this procedure to ansi C and test it there.*

For example, suppose the function is $e^x \cos(2\,x)$ and the interval is [-1,1]. Suppose the error that will be tolerated is $10^{(-5)}$.

```
restart;
```

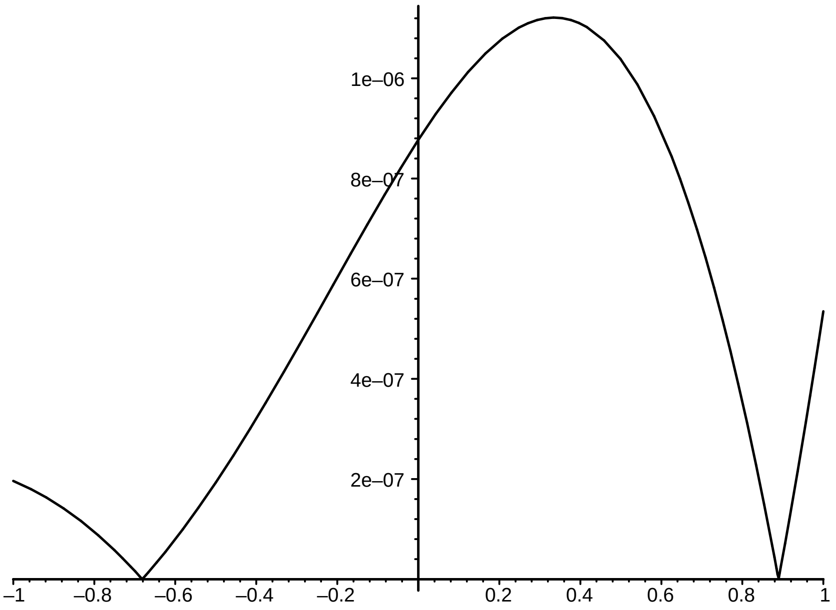
Analysis: Define the function.

```
f := x -> exp(x)*cos(2*x);
```

$$f := x \to e^x \cos(2\,x)$$

By Taylor's theorem, the error in using the nth taylor polynomial is no more than $\frac{M}{(n+1)!}$, where M is an upper bound on the n+1st derivative and 1 is the radius of the interval. We can bound the error in using the nth Taylor polynomial by plotting the function $\frac{\mathrm{D}^{(n+1)}}{(n+1)!}$ over the interval [-1..1] for various n until we find one with a small enough error.

```
plot({0,abs@(D@@14)(f)/14!
},-1..1,color=black);
```



By experimenting, the 13th taylor polynomial about 0 is almost good enough, but is off in the 3rd decimal place near -1. The 12th taylor polynomial is close enough everywhere.

```
p := taylor(f(x),x=0,14);
```

$$p := 1 + x - \frac{3}{2}x^2 - \frac{11}{6}x^3 - \frac{7}{24}x^4 + \frac{41}{120}x^5 + \frac{13}{80}x^6 + \frac{29}{5040}x^7 - \frac{527}{40320}x^8 - \frac{1199}{362880}x^9 + \frac{79}{1209600}x^{10} + \frac{6469}{39916800}x^{11} + \frac{1679}{68428800}x^{12} - \frac{8839}{6227020800}x^{13} + \mathrm{O}(x^{14})$$

```
p :=convert(evalf(p),polynom);
```

$$p := 1. + 1.\,x - 1.500000000\,x^2 - 1.833333333\,x^3 - .2916666667\,x^4 + .3416666667\,x^5 + .1625000000\,x^6 + .005753968254\,x^7 - .01307043651\,x^8 - .003304122575\,x^9 + .00006531084656\,x^{10} + .0001620620891\,x^{11} + .00002453645249\,x^{12} - .1419458885\,10^{-5}\,x^{13}$$

We should convert this to the nested form to make evaluation more efficient.

```
p := convert(p,horner);
```

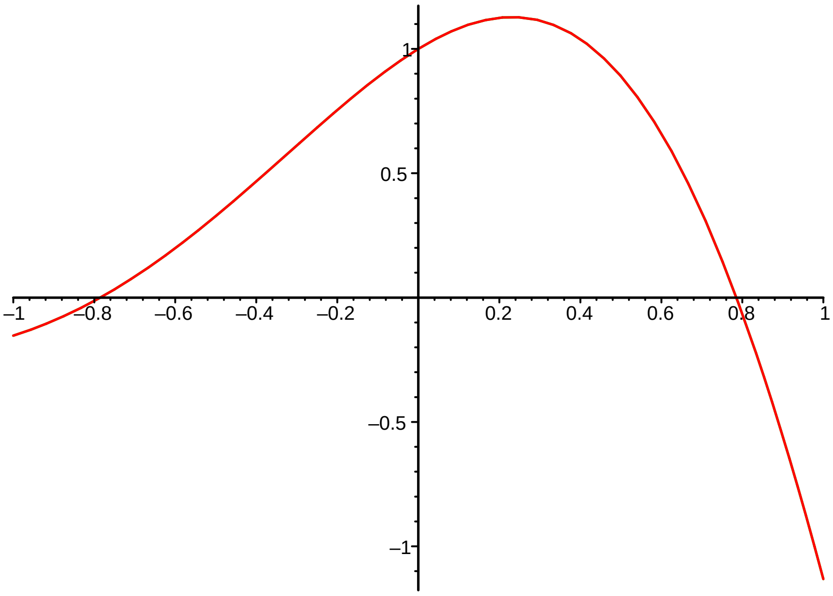
$$p := 1. + (1. + (-1.500000000 + (-1.833333333 + (-.2916666667 + (.3416666667 + (.1625000000 + (.005753968254 + (-.01307043651 + (-.003304122575 + (.00006531084656 + (.0001620620891 + (.00002453645249 - .1419458885\,10^{-5}\,x)\,x)\,x)x)x)x)x)x)x)x)x)x)x$$

```
p := unapply(p,x);
```

$$p := x \to 1. + (1. + (-1.500000000 + (-1.833333333 + (-.2916666667 + (.3416666667 + (.1625000000 + (.005753968254 + (-.01307043651 + (-.003304122575 + (.00006531084656 + (.0001620620891 + (.00002453645249 - .1419458885\,10^{-5}\,x)\,x)\,x)x)x)x)x)x)x)x)x)x)x$$

We can do a quick graphical check to see that the approximating polynomial 'looks like' the original function over the interval.

```
plot({f,p},-1..1);
```



But to really be sure, we should tabulate.

```
printf(‘%6s %16s %16s \n‘, ‘x‘, ‘f(x)‘,
‘p(x)‘);
for i from -10 to 10 do
printf(‘% 6.2e % 15.6e % 15.6e \n‘, i/10.,f(i/10.),p(i/10.))
od;
```

```
     x              f(x)              p(x)
-1.00e+00    -1.530919e-01    -1.530911e-01
-9.00e-01    -9.237348e-02    -9.237330e-02
-8.00e-01    -1.312019e-02    -1.312016e-02
-7.00e-01     8.440319e-02     8.440319e-02
-6.00e-01     1.988662e-01     1.988662e-01
-5.00e-01     3.277099e-01     3.277099e-01
-4.00e-01     4.670165e-01     4.670165e-01
-3.00e-01     6.114237e-01     6.114237e-01
-2.00e-01     7.541010e-01     7.541010e-01
-1.00e-01     8.868009e-01     8.868009e-01
 0.00e-01     1.000000e+00     1.000000e+00
 1.00e-01     1.083141e+00     1.083141e+00
 2.00e-01     1.124986e+00     1.124986e+00
 3.00e-01     1.114087e+00     1.114087e+00
 4.00e-01     1.039364e+00     1.039364e+00
 5.00e-01     8.908079e-01     8.908079e-01
 6.00e-01     6.602589e-01     6.602589e-01
 7.00e-01     3.422718e-01     3.422718e-01
 8.00e-01    -6.498473e-02    -6.498469e-02
 9.00e-01    -5.588270e-01    -5.588268e-01
```

```
 1.00e+00    -1.131204e+00    -1.131203e+00
```

2.2 Exporting to C

Here's a way to get some C++ code developed

Read in the translator.

```
with(codegen);
```

```
Warning, new definition for fortran
```

[*C*, *GRAD*, *GRADIENT*, *HESSIAN*, *JACOBIAN*, *cost*, *declare*, *dontreturn*, *fortran*, *horner*, *intrep2maple*, *joinprocs*, *makeglobal*, *makeparam*, *makeproc*, *makevoid*, *maple2intrep*, *optimize*, *packargs*, *packlocals*, *packparams*, *prep2trans*, *renamevar*, *split*, *swapargs*]

Export f and p to ansi C.

```
C(f,ansi);
```

```
/* The options were    : operatorarrow */
#include <math.h>
double f(double x)
{
  {
    return(exp(x)*cos(2.0*x));
  }
}
```

```
C(p,ansi);
```

```
/* The options were    : operatorarrow */
double p(double x)
{
  {

return(1.0+(1.0+(-0.15E1+(-0.1833333333E1+(-0.2916666667+(0.3416666667
+(
0.1625+(0.5753968254E-2+(-0.1307043651E-1+(-0.3304122575E-2+(0.6531084
656E-4+(
0.1620620891E-3+(0.2453645249E-4-0.1419458885E-5*x)*x)*x)*x)*x)*x)*x)*
x)*x)*x)*
x)*x)*x);
  }
}
```

In order to compare these functions we need a preamble with directives and declarations. Also, we need a main() .

Here is a main.

```
main := proc()
local i;
printf(‘%6s %16s %16s \n‘, ‘x‘, ‘f(x)‘, ‘p(x)‘);
for i from -10 to 10 do
printf(‘% 6.2e % 15.6e % 15.6e \n‘, i/10.,f(i/10.),p(i/10.))  od;
NULL;
end;
```

```
    main := proc()
        local i;
            printf(‘%6s  %16s  %16s \n‘, x, ‘f(x)‘, ‘p(x)‘);
            for i from − 10 to 10 doprintf(‘% 6.2e  % 15.6e  % 15.6e \n‘, .1000000000 × i,
                f(.1000000000 × i), p(.1000000000 × i))
            od;
            NULL
        end
```

```
main();
```

```
     x              f(x)              p(x)
-1.00e+00    -1.530919e-01    -1.530911e-01
-9.00e-01    -9.237348e-02    -9.237330e-02
-8.00e-01    -1.312019e-02    -1.312016e-02
-7.00e-01     8.440319e-02     8.440319e-02
-6.00e-01     1.988662e-01     1.988662e-01
-5.00e-01     3.277099e-01     3.277099e-01
-4.00e-01     4.670165e-01     4.670165e-01
-3.00e-01     6.114237e-01     6.114237e-01
-2.00e-01     7.541010e-01     7.541010e-01
-1.00e-01     8.868009e-01     8.868009e-01
 0.00e-01     1.000000e+00     1.000000e+00
 1.00e-01     1.083141e+00     1.083141e+00
 2.00e-01     1.124986e+00     1.124986e+00
 3.00e-01     1.114087e+00     1.114087e+00
 4.00e-01     1.039364e+00     1.039364e+00
 5.00e-01     8.908079e-01     8.908079e-01
 6.00e-01     6.602589e-01     6.602589e-01
 7.00e-01     3.422718e-01     3.422718e-01
 8.00e-01    -6.498473e-02    -6.498469e-02
 9.00e-01    -5.588270e-01    -5.588268e-01
 1.00e+00    -1.131204e+00    -1.131203e+00
```

```
C(main,ansi);
```

```
#include <math.h>
void main(void)
{
  int i;
  {
    printf( "%6s  %16s  %16s
" ,x,f(x),p(x));
    for(i = -10;i <= 10.0;i++)
C/expression:   Unknown function:   f   will be left as is.
C/expression:   Unknown function:   p   will be left as is.
      printf( "% 6.2e  % 15.6e  % 15.6e
" ,0.1*i,f(0.1*i),p(0.1*i));
    return;
  }
}
```

We print out a complete program that almost works as follows.

```
lprint(‘#include <iostream.h>‘);
lprint(‘#include <math.h>‘);
lprint(‘\n‘);
C(f,ansi);
C(p,ansi);
C(main,ansi);
```

```
#include <iostream.h>
#include <math.h>

/* The options were    : operatorarrow */
#include <math.h>
double f(double x)
{
  {
    return(exp(x)*cos(2.0*x));
  }
}
/* The options were    : operatorarrow */
double p(double x)
{
  {

return(1.0+(1.0+(-0.15E1+(-0.1833333333E1+(-0.2916666667+(0.3416666667
+(
0.1625+(0.5753968254E-2+(-0.1307043651E-1+(-0.3304122575E-2+(0.6531084
656E-4+(
0.1620620891E-3+(0.2453645249E-4-0.1419458885E-5*x)*x)*x)*x)*x)*x)*x)*
x)*x)*x)*
x)*x)*x);
  }
}
#include <math.h>
void main(void)
{
  int i;
  {
    printf( "%6s  %16s  %16s
" ,x,f(x),p(x));
    for(i = -10;i <= 10.0;i++)
      printf( "% 6.2e  % 15.6e  % 15.6e
" ,0.1*i,f(0.1*i),p(0.1*i));
    return;
  }
}
```

The idea is to get something which you can just color and copy into a text file which you can feed to g++.

Note: As it stands, this won't compile. But with a little touch up it will. In the first printf statement in main(), replace the string x,f(x),p(x) with "x", "f(x)", "p(x)" and also just under the declaration int i in main() insert the declaration double x; Now copy and paste it and see if it won't compile.

2.2.1 ***Problem***. *Another way to approximate a function on an interval is with an interpolation polynomial. This has advantages over a Taylor polynomial: No derivatives have to be computed. Tabulate the function above over -1 to 1 with 10 equally spaced nodes and interpolate with a 9th degree polynomial. (Use interp) to generate the polynomial in Maple. What is the error in the approximation?*

3 Romberg integration - More exporting experiments.

We know about the trapezoid rule T_n. It's obtained by integrating the linear spline which interpolates

the data $[x_i, \mathrm{f}(x_i)]$, $i = 0, 1, \ldots, n$, where $x_i = a + \frac{i\,(b-a)}{n}$. Here is

3.1 An efficient sequence of trapezoids

One can see upon inspection that the sequence of trapezoid rules T_{2^n} can be computed in a very efficient manner. Thus $T_2 = \frac{(b-a)\,(\mathrm{f}(a)+2\,\mathrm{f}(a+\frac{b-a}{2}+\mathrm{f}(b)))}{4}$ which after rearangement is $\frac{T_1}{2} + \frac{(b-a)\,\mathrm{f}(a+\frac{b-a}{2})}{2}$. Similarly, $T_4 = \frac{1\,T_2}{2} + \frac{(b-a)\,(\mathrm{f}(a+\frac{b-a}{4})+\mathrm{f}(a+\frac{3\,(b-a)}{4}))}{4}$. This continues on, so that T_{2^n} can be

computed from $T_{2^{(n-1)}}$ and the values of f at the midpoints of the $2^{(n-1)}$ subintervals used in $T_{2^{(n-1)}}$. Here is a Maple word (trap) to compute the terms of the sequence. Note that it is recursively defined.

```
trap := proc(f,a,b,n)
options remember;
local i,h;
h := evalf(b-a):
if n = 1 then evalf(h/2*(f(a)+f(b))) else
evalf( (.5*trap(f,a,b,n-1) +
h/2^(n-1)*sum(f(a+(2.*i-1)*h/2^(n-1)),i=1..2^(n-2)) )) fi end:
for i from 1 to 6 do
2^i,trap(x->exp(x^2),0,1,i) od;
```

```
 2, 1.859140914
 4, 1.571583166
 8, 1.490678862
16, 1.469712277
32, 1.464420310
64, 1.463094103
```

3.2 The Romberg Triangle

For functions analytic on [a,b], the error in using the trapezoid rule to estimate the integral can be shown to be a power series in the even powers of h, the width of the bases of the trapezoids used in the rule. Once one has this fact, one can do the same trick that was used to accelerate the computation of numerical derivatives, to accelerate the computation of the integral. It is called Romberg integration, and it fills in a triangle. The Romberg triangle looks like this

$$\begin{bmatrix} R_{1,1} & & & \\ R_{2,1} & \frac{4\,R_{2,1}}{3} - \frac{1\,R_{1,1}}{3} & & \\ R_{3,1} & \frac{4\,R_{3,1}}{3} - \frac{1\,R_{2,1}}{3} & \frac{16\,R_{3,2}}{15} - \frac{1\,R_{2,2}}{15} & \\ R_{4,1} & \frac{4\,R_{4,1}}{3} - \frac{1\,R_{3,1}}{3} & \frac{16\,R_{4,2}}{15} - \frac{1\,R_{3,2}}{15} & \frac{64\,R_{4,3}}{63} - \frac{1\,R_{3,3}}{63} \end{bmatrix}$$

where the entries down the left column are T_1, T_2, T_4, etc.

Here is a Maple word (romberg) to compute a Romberg triangle.

```
romberg := proc(f,a,b,n)
```

```
local rom,i,j;
Digits := Digits+3;
rom := array(1..n,1..n);
for i from 1 to n do
rom[i,1] := trap(f,a,b,i) od:
for j from 2 to n do for i from j to n do
rom[i,j] := evalf((4^(j-1)*rom[i,j-1]-rom[i-1,j-1])/(4^(j-1)-1))
od; od; evalf(op(rom),Digits-3);
end:
```

Test this out on exp(xˆ2) from 0 to 1/2.

```
romberg(x->exp(x^2),0,.5,5);
```

$$\begin{bmatrix} .5710063542 & ?_{1,2} & ?_{1,3} & ?_{1,4} & ?_{1,5} \\ .5516267918 & .5451669377 & ?_{2,3} & ?_{2,4} & ?_{2,5} \\ .5466559776 & .5449990395 & .5449878463 & ?_{3,4} & ?_{3,5} \\ .5454048909 & .5449878620 & .5449871169 & .5449871053 & ?_{4,5} \\ .5450915865 & .5449871517 & .5449871044 & .5449871042 & .5449871042 \end{bmatrix}$$

```
evalf(evalf(int(exp(x^2),x=0..
.5),13),10);
```

$$.5449871042$$

Upon inspection, we see that convergence is obtained in the 5th row of the triangle.

3.3 Exporting romberg to C++

```
missing operator or `;`
```

When we export romberg to C++, we can store the entries in the table in a matrix. We could use an ordinary 2 dimensional array, however this is a good place to introduce the matrix and vector

classes defined by Pozo. He has provided a collection of basic functions using these classes in a library called libmv.a. These classes and the library provide an extremely convenient way to use matrices and vectors in C++ and we will make extensive use of it as we go along.

Here is a preamble will print the include statements, etc. Note the two includes of vecdefs.h and MATRIX_H . These files are located in the include directory of the development directory that Pozo used to develop the library.

```
preamble := proc()
lprint(` `);
lprint(`#include<stdlib.h>`);
lprint(`#include<stdio.h>`);
lprint(`#include<math.h>`);
lprint(`#include<iostream.h>`);
lprint(`#include "vecdefs.h"`);
lprint(`#include MATRIX_H`);
lprint(` `);
end:
```

Here are slightly modified versions of trap and romberg. Actually, the sum in trap has been rewritten directly. We have inserted declarations into both the procedures, which are ignored by Maple and exported neutrally to C by codegen.

```
trap := proc(f,a,b,n )
```

```
options remember;
local i,h,s;
`double trap(double(double),double,double,int);`;
h := (b-a):
if n = 1 then (h/2*(f(a)+f(b))) else
s:=0;
for i from 1 to 2^(n-2) do
s := s+ f(a+(2*i-1)*h/2^(n-1)) od;
s := h/2^(n-1)*s+.5*trap(f,a,b,n-1);
fi end:
romberg := proc(f,a,b,n)
local rom, i,j;
#rom := array(1..n,1..n);
`MATRIX_double rom(n,n) ;`;
`trap(double(double),double,double,int);`;
for i from 1 to n do
rom[i,1] := trap(f,a,b,i) od:
for j from 2 to n do for i from j to n do
rom[i,j] := ((4^(j-1)*rom[i,j-1]-rom[i-1,j-1])/(4^(j-1)-1)) od;
od; evalm(rom) ;
end:
```

Here is a main which can be used to test our versions of trapezoid and romberg. The declarations in back quotes are ignored by maple. We are experimenting here, and won't be using this method of exporting in future sections.

```
g := x->exp(x^2):
main := proc()
local c, d, g,m, l;
`double trap(double(double),double,double,int);`;
`double romberg(double(double),double,double,int);`;
c := 0;
d := .5:
m := 5:
l := trap(g,c,d,m);
print(l);
romberg(g,c,d,m);
end:
main();
```

```
.03125000000 g(.03125000000) + .03125000000 g(.09375000000)
    + .03125000000 g(.1562500000) + .03125000000 g(.2187500000)
    + .03125000000 g(.2812500000) + .03125000000 g(.3437500000)
    + .03125000000 g(.4062500000) + .03125000000 g(.4687500000)
    + .03125000000 g(.06250000000) + .03125000000 g(.1875000000)
    + .03125000000 g(.3125000000) + .03125000000 g(.4375000000)
    + .03125000000 g(.1250000000) + .03125000000 g(.3750000000)
    + .03125000000 g(.2500000000) + .01562500000 g(0) + .01562500000 g(.5)
                                 rom
```

Now here is a way to get a C++ program that will almost compile.

```
with(codegen):
```

```
Warning, new definition for C
```

```
   preamble();
   C(trap,ansi);
   C(romberg,ansi);
   C(g,ansi);
   C(main,ansi);
```

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<iostream.h>
#include "vecdefs.h"
#include MATRIX_H

/* The options were    : remember */
#include <math.h>
double trap(double f,double a,double b,double n)
{
  int i;
  double h;
  double s;
  {
    double trap(double(double),double,double,int);;
    h = b-a;
    if( n == 1.0 )
      return(h*(f(a)+f(b))/2.0);
    else
      {
        s = 0.0;
        for(i = 1;i <= pow(2.0,1.0*n-2.0);i++)
          s += f(a+(2.0*i-1.0)*h/pow(2.0,1.0*n-1.0));
        s = h/pow(2.0,1.0*n-1.0)*s+0.5*trap(f,a,b,n-1.0);
        return(s);
      }
  }
}
C/pretreatment:   WARNING: Unable to determine array bounds in
rom[i,1]
C/pretreatment:   WARNING: Unable to determine array bounds in
rom[i,j]
C/pretreatment:   WARNING: Unable to determine array bounds in
rom[i,j-1]
C/pretreatment:   WARNING: Unable to determine array bounds in
rom[i-1,j-1]
#include <math.h>
double romberg(double f,double a,double b,double n)
{
  double rom;
  double i;
  int j;
  {
    MATRIX_double rom(n,n) ;;
    trap(double(double),double,double,int);;
    for(i = 1.0;i <= n;i++)
C/expression:   Unknown function:   trap   will be left as is.
      rom[i-1][0] = trap(f,a,b,i);
    for(j = 2;j <= n;j++)
```

```
      for(i = j;i <= n;i++)
        rom[i-1][j-1] =
(pow(4.0,1.0*j-1.0)*rom[i-1][j-2]-rom[i-2][j-2])/(pow(
4.0,1.0*j-1.0)-1.0);
    return(evalm(rom));
  }
}
/* The options were    : operatorarrow */
#include <math.h>
double g(double x)
{
  {
    return(exp(x*x));
  }
}
#include <math.h>
double main(void)
{
  int c;
  double d;
  double g;
  int m;
  double l;
  {
    double trap(double(double),double,double,int);;
    double romberg(double(double),double,double,int);;
    c = 0;
    d = 0.5;
    m = 5;
    l = trap(g,c,d,m);
    printf( "%e\n" ,l);
    return(romberg(g,c,d,m));
  }
}
```

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<iostream.h>
#include”vecdefs.h”
#include MATRIX_H
//
//
double trap(double f(double),double a,double b,int n)
{
  int i;
  double h;
  double s;
  {
   double trap(double(double),double,double,int);;
    h = b-a;
    if( n == 1  )
      return(h*(f(a)+f(b))/2.0);
    else
```

```
      {
        s = 0.0;
        for(i = 1;i <= pow(2.0, n-2 );i++)
          s += f(a+(2* i-1 )*h/pow(2.0,1.0*n-1 ));
        s = h/pow(2.0,1.0*n-1 )*s+0.5*trap(f,a,b,n-1 );
        return(s);
      }
  }
}
//
MATRIX_double romberg(double f(double),double
a,double b,int n)
{
  int i;
  int j;
  {
    MATRIX_double rom(n,n);
    for(i = 1 ;i <= n;i++)
      rom(i-1,0) = trap(f,a,b,i);
    for(j = 2;j <= n;j++)
      for(i = j;i <= n;i++)
        rom(i-1,j-1) = 0.1E1*(pow(4.0,1.0*j-1.0)*rom(i-1
,j-2)-rom(i-2,j-2))/
         (pow(4.0,1.0*j-1.0)-1.0);
    return(rom);
  }
}
//
//
double g(double x)
{
  {
    return(exp(x*x));
  }
}
//
//
void main(void)
{
  int c;
  double d;
  int m;
  double l;
  {
    double trap(double(double),double,double,int);
```

```
    MATRIX_double romberg(double(double),double,double,int);
    c = 0;
    d = 0.5;
    m = 5;
    cout << romberg(g,c,d,m)
  <<endl;
    }
  }
```

The output looks like this

```
  0.571006 -6.52536e-228 3.84067e+06 9.75608e-309 6.59931e+187
  0.551627 0.545167 -3.19062e-294 2.59644e-13 -7.41973e+59
  0.546656 0.544999 0.544988 1.58756e-153 -NaN
  0.545405 0.544988 0.544987 0.544987 9.75609e-309
  0.545092 0.544987 0.544987 0.544987 0.544987
```

3.4 Exporting the erf function.

3.4.1 ***Problem:*** *The function erf is defined by* $\operatorname{erf}(x) = \frac{2\int_0^x e^{(-t^2)}\,dt}{\sqrt{\pi}}$. *Write a procedure using Romberg integration to return its values and export this to C*

A solution: First modify romberg so that it returns the value when two consecutive entries in a row are equal. Also, modify it to compute row by row rather than column by column. (I didn't do this part. How would you do it?)

```
romberg := proc(f,a,b,n)
local rom,i,j;
Digits := Digits+3;
#rom := matrix(n,n,[seq(‘ ‘,i=1..n^2)]);
for i from 1 to n do
rom[i,1] := trap(f,a,b,i) od:
for j from 2 to n do for i from j to n do
rom[i,j] := evalf((4^(j-1)*rom[i,j-1]-rom[i-1,j-1])/(4^(j-1)-1));
if rom[i,j]=rom[i,j-1] then
RETURN(evalf(rom[i,j],Digits-3)) fi;
od; od; op(rom);
end:
```

Test this out on the error function erf.

```
romberg(t->2./evalf(sqrt(Pi))*exp(-t^2),0,.5,8);
```

$$.5204998778$$

```
erf(.5);
```

$$.5204998778$$

Now we can define a myerf.

```
g:=t->2./evalf(sqrt(Pi))*exp(-t^2);
```

$$g := t \to 2.\,\frac{e^{(-t^2)}}{\operatorname{evalf}(\sqrt{\pi})}$$

```
myerf := proc(x)
romberg(g,0,x,8); end;
```

$$\mathit{myerf} := \mathbf{proc}(x)\,\operatorname{romberg}(g,\,0,\,x,\,8)\,\mathbf{end}$$

```
C(romberg,ansi);
C(myerf,ansi);
```

3.4.2 ***Problem.*** *Define a fasterf on -1..1 by 1) tabulating the function at some nodes and then interpolating with a cubic spline. 2) Using a Taylor polynomial. Get 6 digit accuracy. Which is faster?*

3.4.3 ***Problem*** *Invert erf on [-1,1].*

4 Splining data

Interpolation with high degree polynomials is not a good idea in general. Rather than interpolate data with one high degree polynomial, we could interpolate it with continuous piecewise defined functions where the rules are low degree polynomials. These are called splines. The points where the rules change are called knots. The knots are assumed to be listed in increasing order. The x data points are called nodes. Most of the time, knots are nodes. A spline is called a k-spline if it's first k-1 derivatives are continuous, and its k-derivative is a step function, where the 0th derivative is the function itself. Another name for 1-spline (2-spline, 3-spline) is linear spline (quadratic spline, cubic spline). Note that the rules of a k-spline are polynomials of degree <= k.

4.1 The Maple word for splining

There is a nice maple word, *spline*, which can be used to form spline functions. It is not part of the regular vocabulary, but is read in from the library. It takes x and y data, the name of the variable, and the type of spline wanted.

```
readlib(spline);
```

$$\mathbf{proc}(X, Y, z, d)\ \ldots\ \mathbf{end}$$

Using spline returns a set of piecewise rules, which can be fed to unapply to turn them into a piecewise defined function.

```
f :=
unapply(spline([1,2,3],[5,6,3],t,linear),t);
```

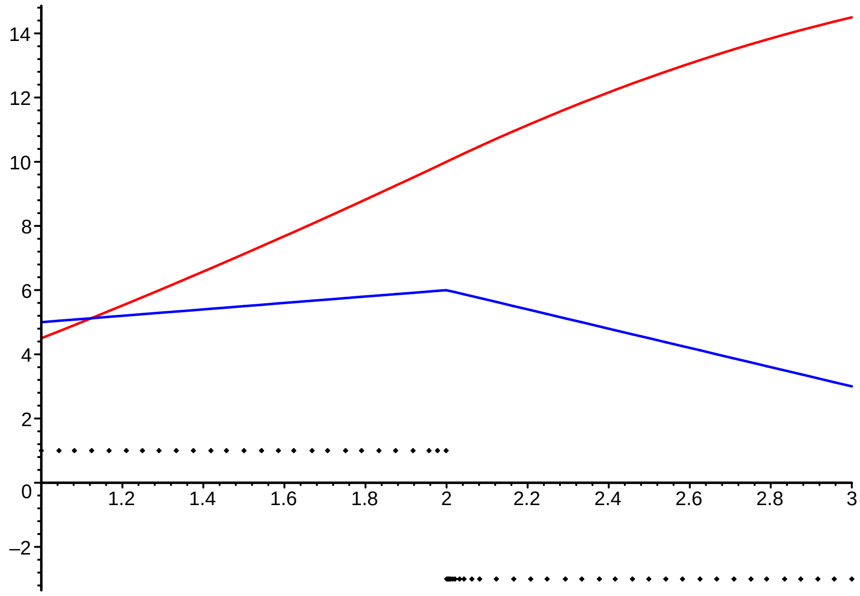
$$f := t \to \mathrm{piecewise}(t < 2,\, 4 + t,\, 12 - 3\,t)$$

The function can then be plotted, differentiated, and integrated among other things.

```
pltf :=
plot(f,1..3,thickness=2,color=blue):
intf := unapply(int(f(t),t),t);
```

$$\mathit{intf} := t \to \mathrm{piecewise}(t \le 2,\, 4\,t + \frac{1}{2}\,t^2,\, 2 < t,\, 12\,t - 8 - \frac{3}{2}\,t^2)$$

```
pltintf :=
plot(intf,1..3,thickness=2,color=red):
pltdf :=
plot(D(f),1..3,style=point,color=black):
plots[display]([pltf,pltintf,pltdf]);
```

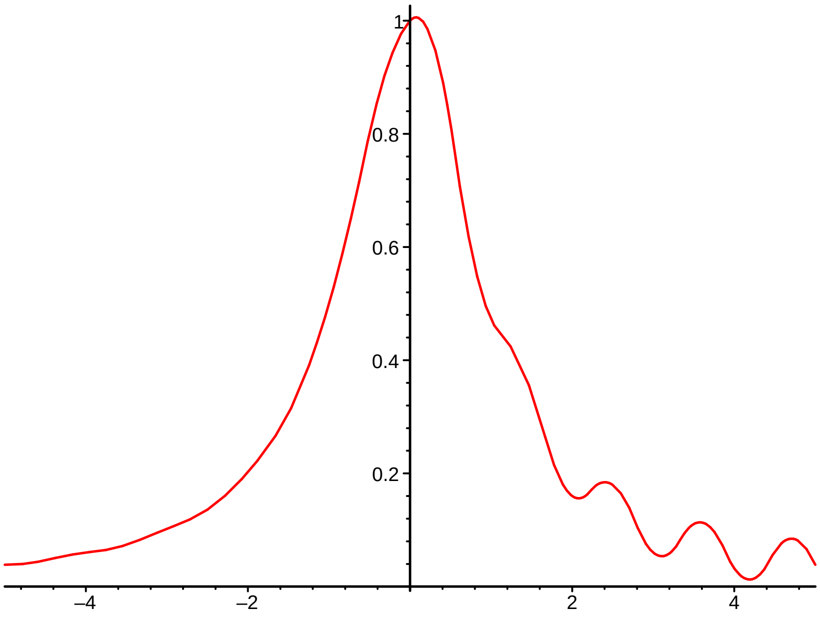


Here is a word which takes a function f, an interval [a,b], the number of knots n, the kind of spline desired and a precision and returns the spline function of the specified kind which interpolates

```
makespline := proc(f,a,b,n,kind)
local x,y,i,t;
x := ([seq(a+i*(b-a)/(n-1),i=0..n-1)]);
y := map(f,x);
readlib(spline);
unapply(spline(x,y,t,kind),t); end:
```

For example, to spline our favorite function with a quadratic spline

```
f :=
makespline(x->1/(1+x^2),-5,5,19,quadratic):
plot({f},-5..5,thickness=2);
```



Here is some strange behaviour. What causes the wiggle? Change the 13 to a 12 and do it again.

4.2 An unnatural quadratic spline

The quadratic spline output by Maple's spline is a 'natural spline', which simply means that its derivative is set to 0 at the leftmost knot. We have seen that this spline is nearly worthless, because of its tendency to 'wiggle' more and more as we go to the left. There is a question as to what causes the wiggle. Is it accumulated floating point error or is it the wrong derivative at the left node? It would be nice to be able to set the derivative to something other than zero.

Here is a word which can be used to construct the quadratic spline in t interpolating data x and y with derivative z1 at x_1.

```
restart;
quadspline := proc(x,y,z1,‘t‘)
local a,b,c,q,i,n,eqns;
n := nops(x);
for i from 1 to n-1 do
q[i] := a[i]*(t-x[i])^2+b[i]*(t-x[i])+c[i]:
q[i] := unapply(q[i],t) od:
eqns := q[n-1](x[n])=y[n], seq(q[i](x[i])=y[i],i=1..n-1):
for i from 1 to n-2 do
eqns := eqns,q[i](x[i+1])=y[i+1],
D(q[i])(x[i+1])=D(q[i+1])(x[i+1]) od:
eqns := eqns,D(q[1])(x[1])=z1:
```

```
assign(solve({eqns},indets({eqns})));
readlib(piecewise);
unapply(piecewise(seq(op([t<x[i+1],q[i](t)]),i=1..n-1),
q[n-1](t)),t);
end:
```

Test this out on our favorite function $\frac{1}{1+t^2}$. (Note we are switching to 16 Digits precision because we are going to export to C later on down.)

```
Digits := 16;
```

$$\mathit{Digits} := 16$$

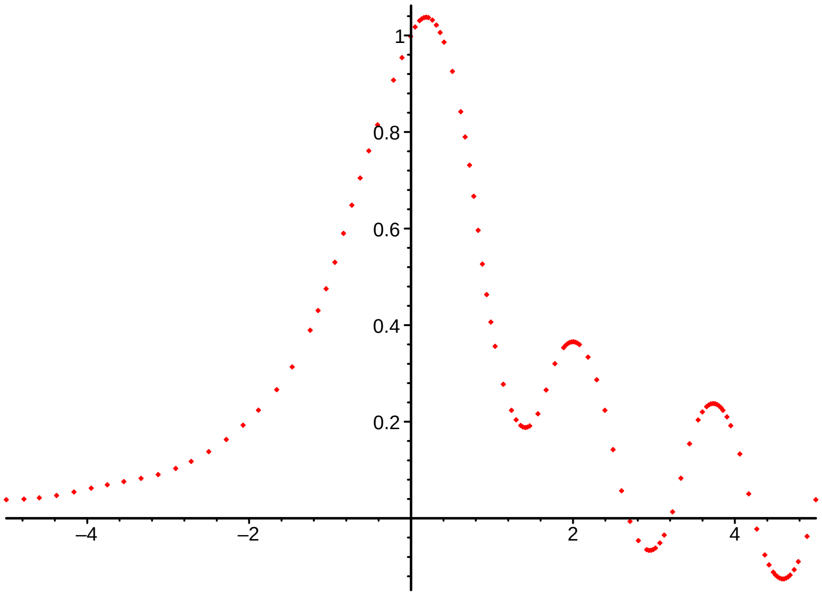
```
x := evalf([seq(-5+i*10/12,i=0..12)]);
```

$$x := [-5., -4.166666666666667, -3.333333333333333, -2.500000000000000, -1.666666666666667, -.8333333333333333, 0, .8333333333333333, 1.666666666666667, 2.500000000000000, 3.333333333333333, 4.166666666666667, 5.]$$

```
y := map(t->1/(1+t^2),x);
```

$$y := [.03846153846153846, .05446293494704993, .08256880733944955, .1379310344827586, .2647058823529411, .5901639344262297, 1, .5901639344262297, .2647058823529411, .1379310344827586, .08256880733944955, .05446293494704993, .03846153846153846]$$

```
f := quadspline(x,y,0,t):
plot(f,-5..5,style=point);
```



Hmmm. Same wiggle.

4.2.1 ***Problem:*** *Experiment with quadspline, with the idea in mind of trying to decide what is responsible for the wiggle. For example, change the value of z1 to 5/338; change the interval to 0 to 10; change the function to* $\frac{1}{1+x}$ *on 0 to 10., to cos on 0 to 2; vary the number of knots; also vary the precision. Make observations and form a conjecture if there is one. There is a 'non natural' quadratic spline described by Y. Subbotin in On Piecewise Polynomial Approximation,* ***Mat. Zametcki*** *1, 63-70, 1967 (or see p. 283 of* ***Numerical Mathematics and Computing****, by Cheney and Kincaid, 3rd Ed. Brooks-Cole). Define a Maple word to generate Subbotin spline functions and test it. Does it behave any better?*

4.3 Exporting a quadratic spline function to C++

4.3.1 ***Problem****: Experiment with exporting splining procedures to C++. To get started, suppose we wanted to export the last quadratic spline we generated above. Then load in the package codegen.*

```
with(codegen);
```

```
Warning, new definition for fortran
        [C, GRAD, GRADIENT, HESSIAN, JACOBIAN, cost, declare, dontreturn, fortran,
            horner, intrep2maple, joinprocs, makeglobal, makeparam, makeproc, makevoid,
            maple2intrep, optimize, packargs, packlocals, packparams, prep2trans,
            renamevar, split, swapargs]
```

Note the word C, which translates C, is defined in this package. So we can try to translate f to ansi C.

```
C(f,ansi);
```

```
/* The options were    : operatorarrow */
#include <math.h>
double f(double t)
{
  {
C/expression:   Unknown function:   piecewise   will be left as is.
    return(piecewise(t <
-0.4166666666666667E1,0.2304201093913653E-1*pow(t+5.0,
2.0)+0.3846153846153846E-1,t <
-0.3333333333333333E1,-0.5611565633217626E-2*pow
(t+0.4166666666666667E1,2.0)+0.3840335156522754E-1*t+0.214476899802164
7,t <
---------- stuff deleted ---------
```

This is not immediately useful, because the word *piecewise* is not a C++ word. We could write a separate C++ procedure with that name and compile it with this procedure to get something useful. A faster way is to convert piecewise to something which can be handled by the existing C++ vocabulary. That is what prep2trans does.

```
f :=prep2trans(f);
```

```
$f :=$ proc($t$)
    option $operator, arrow$;
        if $t < -4.166666666666667$ then
            $.02304201093913653 \times (t + 5.)^2 + .03846153846153846$
        elif $t < -3.333333333333333$ then
            $- .005611565633217626 \times (t + 4.166666666666667)^2 + .03840335156522754 \times t$
            $+ .2144768998021647$
        elif $t < -2.500000000000000$ then
            $.04486071647452729 \times (t + 3.333333333333333)^2 + .02905074217653149 \times t$
            $+ .1794046145945545$
        elif $t < -1.666666666666667$ then
            $.05797345737217056 \times (t + 2.500000000000000)^2 + .1038186029674103 \times t$
            $+ .3974775419012844$
        elif $t < -.8333333333333333$ then
            $.2281303566803019 \times (t + 1.666666666666667)^2 + .2004410319210278 \times t$
            $+ .5987742688879875$
        elif $t < 0$ then $- .1066260172396080 \times (t + .8333333333333333)^2$
            $+ .5806582930548643 \times t + 1.074045845305283$
        elif $t < .8333333333333333$ then
            $- 1.073701851612851 \times t^2 + .4029482643221844 \times t + 1.$
        elif $t < 1.666666666666667$ then
            $1.195206191053544 \times (t - .8333333333333333)^2 - 1.386554821699234 \times t$
            $+ 1.745626285842258$
        elif $t < 2.500000000000000$ then
            $- .9091023770010720 \times (t - 1.666666666666667)^2 + .6054554967233406 \times t$
            $- .7443866121859599$
        elif $t < 3.333333333333333$ then
            $1.011936550847770 \times (t - 2.500000000000000)^2 - .9097151316117787 \times t$
            $+ 2.412218863512206$
        elif $t < 4.166666666666667$ then
            $- .9726874000064589 \times (t - 3.333333333333333)^2 + .7768457864678369 \times t$
            $- 2.506917147553339$
        elif $t < 5.$ then $.9901178453123789 \times (t - 4.166666666666667)^2$
            $- .8442998802095959 \times t + 3.572379102487033$
        else $.9901178453123789 \times (t - 4.166666666666667)^2 - .8442998802095959 \times t$
            $+ 3.572379102487033$
        fi
    end
```

Now we can export this to C. We can make a quick program *main* to test f by tabulating it over the interval 1..3 at .25 intervals

```
main := proc()
local i;
for i from 0 to 12 do
printf(` % 18.15g % 18.15g\n`, 1.0 + i*2./12, f(1.0+i*2./12)) od
end:
```

Test main to get the output.

```
main();
```

```
  1.0                  .392271636116734
  1.166666666666667    .260779681754657
  1.333333333333333    .195688071339999
  1.5                  .19699680487276
  1.666666666666667    .264705882352941
  1.833333333333334    .340362288001247
  2                    .36551300603838
  2.166666666666667    .340158036464344
  2.333333333333334    .264297379279136
  2.5                  .137931034482759
  2.666666666666667    .014421194515456
  2.833333333333334   -.052869948182526
  3                   -.063942393611188
```

Now export to ansi C.

```
C(f,ansi);
C(main,ansi);
```

```
/* The options were    : operatorarrow */
#include <math.h>
double f(double t)
{
  {
    if( t < -0.4166666666666667E1 )

return(0.2304201093913653E-1*pow(t+5.0,2.0)+0.3846153846153846E-1);
    else
      if( t < -0.3333333333333333E1 )
        return(-0.5611565633217626E-2*pow(t+0.4166666666666667E1,2.0)+
0.3840335156522754E-1*t+0.2144768998021647);
      else
        if( t < -0.25E1 )

return(0.4486071647452729E-1*pow(t+0.3333333333333333E1,2.0)+
0.2905074217653149E-1*t+0.1794046145945545);
        else
          if( t < -0.1666666666666667E1 )

return(0.5797345737217056E-1*pow(t+0.25E1,2.0)+0.1038186029674103*t
+0.3974775419012844);
          else
            if( t < -0.8333333333333333 )

return(0.2281303566803019*pow(t+0.1666666666666667E1,2.0)+
0.2004410319210278*t+0.5987742688879875);
            else
              if( t < 0.0 )

return(-0.106626017239608*pow(t+0.8333333333333333,2.0)+
0.5806582930548643*t+0.1074045845305283E1);
              else
                if( t < 0.8333333333333333 )

return(-0.1073701851612851E1*t*t+0.4029482643221844*t+1.0);
                else
                  if( t < 0.1666666666666667E1 )

return(0.1195206191053544E1*pow(t-0.8333333333333333,2.0)
-0.1386554821699234E1*t+0.1745626285842258E1);
```

```
         else
          if( t < 0.25E1 )

return(-0.909102377001072*pow(t-0.1666666666666667E1,2.0)
+0.6054554967233406*t-0.7443866121859599);
          else
           if( t < 0.3333333333333333E1 )
            return(0.101193655084777E1*pow(t-0.25E1,2.0)
-0.9097151316117787*t+0.2412218863512206E1);
           else
            if( t < 0.4166666666666667E1 )

return(-0.9726874000064589*pow(t-0.3333333333333333E1
,2.0)+0.7768457864678369*t-0.2506917147553339E1);
            else
             if( t < 5.0 )
              return(0.9901178453123789*pow(t
-0.4166666666666667E1,2.0)-0.8442998802095959*t+0.3572379102487033E1);
             else
              return(0.9901178453123789*pow(t
-0.4166666666666667E1,2.0)-0.8442998802095959*t+0.3572379102487033E1);
 }
}
#include <math.h>
void main(void)
{
 int i;
 {
  for(i = 0;i <= 12.0;i++)
C/expression:  Unknown function:  f  will be left as is.
   printf( " % 18.15g   % 18.15g
" ,0.1E1+0.1666666666666667*i,f(0.1E1+
0.1666666666666667*i));
 }
}
```

After we paste this into an editor, we have to modify it a little to get it to compiler in the gnu compiler. Specifically, insert an #include <stdio.h> after the first #include <math.h>, and remove the line starting with C/expression .... Also insert a carriage return \n in front of the second " in the printf statement. You can also remove the second #include <math.h>. That is all we need to do for the gnu compiler.

```
/* The options were   : operatorarrow */
#include <math.h>
#include <stdio.h>
double f(double t)
{
 {
  if( t < -0.4166666666666667E1 )

return(0.2304201093913653E-1*pow(t+5.0,2.0)+0.3846153846153846E-1);
  else
   if( t < -0.3333333333333333E1 )
    return(-0.5611565633217626E-2*pow(t+0.4166666666666667E1,2.0)+
0.3840335156522754E-1*t+0.2144768998021647);
```

```
      else
        if( t < -0.25E1 )

return(0.4486071647452729E-1*pow(t+0.3333333333333333E1,2.0)+
0.2905074217653149E-1*t+0.1794046145945545);
        else
          if( t < -0.1666666666666667E1 )

return(0.5797345737217056E-1*pow(t+0.25E1,2.0)+0.1038186029674103*t
+0.3974775419012844);
          else
            if( t < -0.8333333333333333 )

return(0.2281303566803019*pow(t+0.1666666666666667E1,2.0)+
0.2004410319210278*t+0.5987742688879875);
            else
              if( t < 0.0 )

return(-0.106626017239608*pow(t+0.8333333333333333,2.0)+
0.5806582930548643*t+0.1074045845305283E1);
              else
                if( t < 0.8333333333333333 )

return(-0.1073701851612851E1*t*t+0.4029482643221844*t+1.0);
                else
                  if( t < 0.1666666666666667E1 )

return(0.1195206191053544E1*pow(t-0.8333333333333333,2.0)
-0.1386554821699234E1*t+0.1745626285842258E1);
                  else
                    if( t < 0.25E1 )

return(-0.909102377001072*pow(t-0.1666666666666667E1,2.0)
+0.6054554967233406*t-0.7443866121859599);
                    else
                      if( t < 0.3333333333333333E1 )
                        return(0.101193655084777E1*pow(t-0.25E1,2.0)
-0.9097151316117787*t+0.2412218863512206E1);
                      else
                        if( t < 0.4166666666666667E1 )

return(-0.9726874000064589*pow(t-0.3333333333333333E1
,2.0)+0.7768457864678369*t-0.2506917147553339E1);
                        else
                          if( t < 5.0 )
```

```
                              return(0.9901178453123789*pow(t
-0.4166666666666667E1,2.0)-0.8442998802095959*t+0.3572379102487033E1);
                            else
                              return(0.9901178453123789*pow(t
-0.4166666666666667E1,2.0)-0.8442998802095959*t+0.3572379102487033E1);
  }
}
// #include <math.h>
void main(void)
{
  int i;
  {
    for(i = 0;i <= 12.0;i++)
// C/expression:   Unknown function:   f   will be left as is.
      printf( ” % 18.15g    % 18.15g\n”
,0.1E1+0.1666666666666667*i,f(0.1E1+
0.1666666666666667*i));
  }
}
                 1     0.392271636116734
  1.16666666666667     0.260779681754656
  1.33333333333333     0.195688071339999
               1.5      0.19699680487276
  1.66666666666667     0.264705882352941
  1.83333333333333     0.340362288001246
                 2      0.36551300603838
  2.16666666666667     0.340158036464344
  2.33333333333333     0.264297379279136
               2.5     0.137931034482759
  2.66666666666667     0.014421194515456
  2.83333333333333    -0.052869948182526
                 3    -0.063942393611188
```

5 Exporting Cubicspline to C++

We want to show one way to export the cubic spline generating algorithm that we have coded in Maple to C++. Here is the Maple version.

```
cubicspline := proc(x,y,z1,zn)
local i,dx,dy,z,d,r,n,a,b,c,t,s,tmp;
n := nops(x);
z[1]:= z1:  z[n]:=zn:
for i from 1 to n-1 do
dx[i]:=x[i+1]-x[i]:
dy[i]:=y[i+1]-y[i] od:
for i from 1 to n-2 do d[i]:= 2*(dx[i]+dx[i+1]):
r[i]:= 6*(dy[i+1]/dx[i+1]-dy[i]/dx[i]) od:
r[1] := r[1]-dx[1]*z[1]:
for i from 2 to n-2 do
d[i] := d[i]-dx[i+1]^2/d[i-1]:
r[i] := r[i]-dx[i]*r[i-1]/d[i-1]
od:
for i from n-1 by -1 to 2 do
z[i] := (r[i-1]-dx[i]*z[i+1])/d[i-1] od:
for i from 1 to n-1 do
a[i] := 1/6*(z[i+1]-z[i])/dx[i]:
b[i] := z[i]/2:
c[i] := dy[i]/dx[i]-a[i]*dx[i]^2-b[i]*dx[i] :
s[i] := a[i]*(t-x[i])^3+b[i]*(t-x[i])^2+c[i]*(t-x[i])+y[i]:
s[i] := unapply(s[i],t) od:
readlib(piecewise);
unapply(piecewise(seq(op([t<x[i+1],s[i](t)]),i=1..n-1),
s[n-1](t)),t);
end:
```

We will break this into two tasks. First, we need code to return the coefficients of the rules. Second we need code to evaluate the spline function at a given x. So, we load the codegen package and use the C word on cubicspline after modifying it to return the coefficients of the cubic spline which interpolates the data x, y.

```
cubicspline := proc(x,y,z1,zn)
local i,dx,dy,z,d,r,a,b,c,n,t,cofs;
n := nops(x);
z[1]:= z1:  z[n]:=zn:
for i from 1 to n-1 do
dx[i]:=x[i+1]-x[i]:
dy[i]:=y[i+1]-y[i] od:
for i from 1 to n-2 do d[i]:= 2*(dx[i]+dx[i+1]):
r[i]:= 6*(dy[i+1]/dx[i+1]-dy[i]/dx[i]) od:
r[1] := r[1]-dx[1]*z[1]:
for i from 2 to n-2 do
d[i] := d[i]-dx[i+1]^2/d[i-1]:
r[i] := r[i]-dx[i]*r[i-1]/d[i-1]
od:
for i from n-1 by -1 to 2 do
z[i] := (r[i-1]-dx[i]*z[i+1])/d[i-1] od:
for i from 1 to n-1 do
```

```
a[i] := 1/6*(z[i+1]-z[i])/dx[i]:
b[i] := z[i]/2:
c[i] := dy[i]/dx[i]-a[i]*dx[i]^2-b[i]*dx[i] :
od:
for i from 1 to n-1 do
cofs[i,1]:=a[i]:  cofs[i,2]:=b[i]:
cofs[i,3]:= c[i]:  cofs[i,4]:=y[i]:  od:
cofs;
end:
with(codegen);
```

```
Warning, new definition for C
       [C, GRAD, GRADIENT, HESSIAN, JACOBIAN, cost, declare, dontreturn, fortran,
          horner, intrep2maple, joinprocs, makeglobal, makeparam, makeproc, makevoid,
          maple2intrep, optimize, packargs, packlocals, packparams, prep2trans,
          renamevar, split, swapargs]
```

Note: There are loads of warning messages about not being able to determine arraybounds for various things. Pay no heed! Just color all of the output and, using the mouse change the paragraph style (in the context bar above) . Then you can modify the exported code and copy it to a separate file for compilation. We will need to do some modification of the exported code. Namely, we need to declare the vectors and matrices used in the code using the types VECTOR_double and MATRIX_double defined in Pozo's MV. We also add two include statements for the headers accompanying his library.

```
C(cubicspline,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in   dx[i]
C/pretreatment:   WARNING: Unable to determine array bounds in   a[i]
C/pretreatment:   WARNING: Unable to determine array bounds in   dx[i]
C/pretreatment:   WARNING: Unable to determine array bounds in   b[i]
C/pretreatment:   WARNING: Unable to determine array bounds in   dx[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,1]
C/pretreatment:   WARNING: Unable to determine array bounds in   a[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,2]
C/pretreatment:   WARNING: Unable to determine array bounds in   b[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,3]
C/pretreatment:   WARNING: Unable to determine array bounds in   c[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,4]
C/pretreatment:   WARNING: Unable to determine array bounds in   y[i]
#include <math.h>
double cubicspline(double x,double y,double z1,double zn)
{
  int i;
  double dx;
  double dy;
  double z;
  double d;
  double r;
  double a;
  double b;
  double c;
  int n;
```

```
  double t;
  double cofs;
  {
    n = nops(x);
    z[0] = z1;
    z[n-1] = zn;
    for(i = 1;i <= n-1.0;i++)
    {
      dx[i-1] = x[i]-x[i-1];
      dy[i-1] = y[i]-y[i-1];
    }
    for(i = 1;i <= n-2.0;i++)
    {
      d[i-1] = 2.0*dx[i-1]+2.0*dx[i];
      r[i-1] = 6.0*dy[i]/dx[i]-6.0*dy[i-1]/dx[i-1];
    }
    r[0] += -dx[0]*z[0];
    for(i = 2;i <= n-2.0;i++)
    {
      d[i-1] += -dx[i]*dx[i]/d[i-2];
      r[i-1] += -dx[i-1]*r[i-2]/d[i-2];
    }
    for(i = n-1;2.0 <= i;i--)
      z[i-1] = (r[i-2]-dx[i-1]*z[i])/d[i-2];
    for(i = 1;i <= n-1.0;i++)
    {
      a[i-1] = (z[i]-z[i-1])/dx[i-1]/6.0;
      b[i-1] = z[i-1]/2.0;
      c[i-1] = dy[i-1]/dx[i-1]-a[i-1]*pow(dx[i-1],2.0)-b[i-1]*dx[i-1];
    }
    for(i = 1;i <= n-1.0;i++)
    {
      cofs[i-1][0] = a[i-1];
      cofs[i-1][1] = b[i-1];
      cofs[i-1][2] = c[i-1];
      cofs[i-1][3] = y[i-1];
    }
    return(cofs);
  }
}
```

Here is the modified exported code.
// This C++ word takes data x and y and values z1 and zn for the second derivative at the // right hand and left hand knots and returns the coefficients cofs of the cubic spline // which interpolates this data. MATRIX_double cubicspline(VECTOR_double x,VECTOR_double y,double z1,double zn) { int n=x.size(); VECTOR_double dx(n,0); VECTOR_double dy(n,0); VECTOR_double z(n,0); VECTOR_double d(n-1,0); VECTOR_double r(n,0); VECTOR_double a(n-1,0) ; VECTOR_double b(n-1,0) ; VECTOR_double c(n-1,0) ; MATRIX_double cofs(n-1,4); double t; { int i =0; z(0) = z1; z(n-1) = zn; for(i = 1;i <= n-1.0;i++) { dx(i-1) = x(i)-x(i-1); dy(i-1) = y(i)-y(i-1); } for(i = 1;i <= n-2.0;i++) { d(i-1) = 2.0*dx(i-1)+2.0*dx(i); r(i-1) = 6.0*dy(i)/dx(i)-6.0*dy(i-1)/dx(i-1); } r(0) += -dx(0)*z(0); for(i = 2;i <= n-2.0;i++) { d(i-1) += -dx(i)*dx(i)/d(i-2); r(i-1) += -dx(i-1)*r(i-2)/d(i-2); } for(i = n-1;2.0 <= i;i–) z(i-1) = (r(i-2)-dx(i-1)*z(i))/d(i-2); for(i = 1;i <= n-1.0;i++) { a(i-1) = (z(i)-z(i-1))/dx(i-1)/6.0; b(i-1) = z(i-1)/2.0; c(i-1) = dy(i-1)/dx(i-1)-a(i-1)*pow(dx(i-1),2.0)-b(i-1)*dx(i-1);

} for(i=0;i<n-1;i++) cofs(i,0)=a(i); cofs(i,1)=b(i); cofs(i,2)=c(i); cofs(i,3)=y(i); return(cofs); }

Now we need to develop a word to evaluate a cubic spline with knots x and coefficients cofs at a point t. We use a simple left to right search to locate the interval $[x_i, x_{i+1}]$ that t lies in so that the correct rule can be applied.

```
csplfun := proc(x,cofs,t)
local n,i,j;
n := nops(x);
i := n-1;
for j from 1 to n-2 do
if t<=x[j+1] then i := j ; break; fi; od;
RETURN(((cofs[i,1]*(t-x[i])+cofs[i,2])*(t-x[i])+cofs[i,3])*(t-x[i])+co
fs[i,4]) end:
C(csplfun,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
x[j+1]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,1]
C/pretreatment:   WARNING: Unable to determine array bounds in   x[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,2]
C/pretreatment:   WARNING: Unable to determine array bounds in   x[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,3]
C/pretreatment:   WARNING: Unable to determine array bounds in   x[i]
C/pretreatment:   WARNING: Unable to determine array bounds in
cofs[i,4]
#include <math.h>
double csplfun(double x,double cofs,double t)
{
  double n;
  int i;
  int j;
  {
    n = nops(x);
    i = n-1;
    for(j = 1;j <= n-2.0;j++)
      if( t <= x[j] )
        {
          i = j;
          break;
        }

return(((cofs[i-1][0]*(t-x[i-1])+cofs[i-1][1])*(t-x[i-1])+cofs[i-1][2]
)*(t-
x[i-1])+cofs[i-1][3]);
  }
}
```

Now here is a modification that will compile with gcc.

```
#include "vecdefs.h"
#include MATRIX_H
#include <math.h>
//
double csplfun(const VECTOR_double & x,const
MATRIX_double & cofs,double t)
{
```

```
  int n;
  int i;
  int j;
  {
    n = x.size();
    i = n-1;
    for(j = 1;j <= n-2.0;j++)
      if( t <= x(j) )
        {
          i = j;
          break;
        }
return(((cofs(i-1,0)*(t-x(i-1))+cofs(i-1,1))*(t-x(i-1))+cofs(i-1,2))*(
t-
x(i-1))+cofs(i-1,3));
  }
}
```

6 Inverting a function

Suppose f is a continous function for which we have a well-defined way of computing f(x) for x in the interval [a,b]. Suppose further that we know f is 1-1 on that interval. Then of course f has an inverse g defined by g(y) = x means x is in [a,b] and f(x) = y. How can we compute g(y), for y between f(a) and f(b)? Two ways come to mind.

By Inverse interpolation. One way is use inverse interpolation to obtain an approximation to the inverse. This is one use of polynomial interpolation. Tabulate the function f from a to b using some finite number of nodes x. So we have an x table and a corresponding y table. Now just interchange x and y and let p be the polynomial of minimum degree for which p(y_i) = x_i. As long as there is not too much 'wiggle', the polynomial p should be a reasonable approximation to the inverse g.

By solving equations. The most immediate way is to solve the equation f(y) = x for y. Thus we are led to investigate ways to solve equations. Maple has a very good quality equation solver builtin called fsolve. So to start we could just use that word.

```
invert := proc(f,y,a,b)
local x;
fsolve(f(x)=y,x,a..b) end;
```

$$invert := \mathbf{proc}(f,\ y,\ a,\ b)\ \ \mathbf{local}\,x;\ \mathrm{fsolve}(f(x) = y,\ x,\ a..b)\,\mathbf{end}$$

```
invert(exp,1.1,0,2);
```

$$.09531017980$$

If we want to write a procedure which we can export to C, we must write our own fsolve. There are three methods which are especially useful: bisection, newton, and the secant methods.

6.1 Bisection

Bisection: Suppose [a,b] is an interval in the domain of f and f(a) and f(b) have different signs. Then by the intermediate value theorem for continuous functions f(x)=0 has a solution in [a,b]. Let m be the midpoint of the interval. Compare the sign of f(m) with the sign of f(a): if they agree, then update a = m. If they don't then update b = m. In either case we still have a solution of f(x) between a and b and b-a has been cut in half. This can be iterated until the relative error (b-a)/a is as small as we like (in theory, in practice you can't do better than machine epsilon). Here is one implementation of bisection which can be used to invert a function.

```
bisect := proc(f,a1,b1)
local a,b,m,d;
Digits := Digits +3;
a:= evalf(a1);
b:= evalf(b1);
m := (a+b)/2.;
while abs((b-m)/m) > .5*1/10^(Digits-1) do
if f(a)*f(m) <= 0 then b := m else a := m fi;
m := (a+b)/2.;
od;
evalf(m,Digits-3); end:
```

So, to find the square root of 2 for example, we could call bisect with f the function $x^2 - 2$, and a search interval of [1,2].

```
bisect(x->x^2-2,1,2);
```

```
1.414213562373095
```

Here is a modification of invert above to make use of the idea of bisect.

```
invert := proc(f,y,a1,b1)
local m,d,a,b;
Digits := Digits +3;
a:= evalf(a1);
b:= evalf(b1);
m := (a+b)/2.;
while abs((b-m)/m) > .5*1/10^(Digits-1) do
if (f(a)-y)*(f(m)-y) <= 0 then b := m else a := m fi;
m := (a+b)/2.;
od;
evalf(m,Digits-3); end:
invert(x->x^2,2,0,4);
```

```
1.414213562373095
```

This is something we can export to C++and here is a main to go with it. We will invert the squaring function over the interval 1 to 9.

```
f := x -> x^2:
main := proc()
local x;
for x from 1 by 1 to 9 do
print(x,invert(f,x,1,3));
od;
end:
main();
```

```
1, 1.000000000
2, 1.414213562
3, 1.732050808
4, 2.000000000
5, 2.236067978
6, 2.449489743
7, 2.645751311
8, 2.828427125
9, 3.000000000
```

Now convert these procedures to ansi C using the codegen word C

```
codegen[C](bisect,ansi);
codegen[C](f,ansi);
codegen[C](invert,ansi);
codegen[C](main,ansi);
```

```
#include <math.h>
double bisect(double f,double a1,double b1)
{
  double a;
  double b;
  double m;
  double d;
  {
    Digits += 3.0;
    a = 0.1E1*a1;
    b = 0.1E1*b1;
```

```
    m = 0.5*a+0.5*b;
    while(0.5/pow(10.0,1.0*Digits-1.0) < fabs((b-m)/m))
    {
      if( f(a)*f(m) <= 0.0 )
        b = m;
      else
        a = m;
      m = 0.5*a+0.5*b;
    }
    return(0.1E1*m);
  }
}
/* The options were    : operatorarrow */
double f(double x)
{
  {
    return(x*x);
  }
}
#include <math.h>
double invert(double f,double y,double a1,double b1)
{
  double m;
  double d;
  double a;
  double b;
  {
    Digits += 3.0;
    a = 0.1E1*a1;
    b = 0.1E1*b1;
    m = 0.5*a+0.5*b;
    while(0.5/pow(10.0,1.0*Digits-1.0) < fabs((b-m)/m))
    {
      if( (f(a)-y)*(f(m)-y) <= 0.0 )
        b = m;
      else
        a = m;
      m = 0.5*a+0.5*b;
    }
    return(0.1E1*m);
  }
}
#include <math.h>
void main(void)
{
  double x;
  {
    for(x = 1.0;x <= 3.0;x += 0.25)
C/expression:   Unknown function:   invert   will be left as is.
      printf( "%e%e\n" ,x,invert(f,x,1.0,9.0));
  }
}
```

Here is a modifid version which compiles and runs with gcc. Note: inputs which are functions must be cast.

```
#include <math.h>
#include <stdio.h>
//
double bisect(double f(double),double a1,double b1)
```

```
{
  double a;
  double b;
  double m;
  double d;
  {
    a = 0.1E1*a1;
    b = 0.1E1*b1;
    m = 0.5*a+0.5*b;
    while(0.5/pow(10.0,1.0*-10) < fabs((b-m)/m))
    {
      if( f(a)*f(m) <= 0.0 )
        b = m;
      else
        a = m;
      m = 0.5*a+0.5*b;
    }
    return(0.1E1*m);
  }
}
//
double f(double x)
{
  {
    return(x*x);
  }
}
//
double invert(double f(double),double y,double a1,double b1)
{
  double m;
  double d;
  double a;
  double b;
  {
    a = 0.1E1*a1;
    b = 0.1E1*b1;
    m = 0.5*a+0.5*b;
    while(0.5/pow(10.0,1.0*10) < fabs((b-m)/m))
    {
      if( (f(a)-y)*(f(m)-y) <= 0.0 )
        b = m;
      else
        a = m;
      m = 0.5*a+0.5*b;
```

```
    }
    return(0.1E1*m);
  }
}
//
void main(void)
{
  double x;
  {
    for(x = 1.0;x <= 9.0;x +=1)
      printf( "%e   %e\n"
,x,invert(f,x,1.0,3.0));
  }
}
```

Save this as bisect.cc. Assuming you are working in a Tex Shell, compile it with the command

```
gcc -c bisect.cc
```

Then create an execuatble with the command

```
gcc -o invert bisect.o -lm -lgpp
```

Then execute invert by the command

```
invert
```

The output should look like this.

```
1.000000e+00   1.000000e+00
2.000000e+00   1.414214e+00
3.000000e+00   1.732051e+00
4.000000e+00   2.000000e+00
5.000000e+00   2.236068e+00
6.000000e+00   2.449490e+00
7.000000e+00   2.645751e+00
8.000000e+00   2.828427e+00
9.000000e+00   3.000000e+00
```

6.2 Newton's method

Newton's Method: Suppose f is differentiable and x1 is a first approximation to a solution r of the equation f(x)=0. Newton say's find the root x2 of the tangent line approximation to f at x1, that ought to be closer to the root r we seek. This amounts to solving the equation $f'(x1) = \frac{0-\mathrm{f}(x1)}{x2-x1}$ for x2. Get $x2 = x1 - \frac{\mathrm{f}(x1)}{f'(x1)}$. This is iterated until, hopefully, convergence is obtained. Since we don't know r, we can only estimate the relative error in using x1 to approximate r with $\left|\frac{x2-x1}{x2}\right|$. Here is an implementation of Newton's method in Maple. All we need to get started is f, and a first approximation to the root.

```
newton:=proc(f,r1)
local df,x1,x2,i;
Digits := Digits+3;
df := D(f);
x1 := evalf(r1);
x2 := x1 - evalf(f(x1)/df(x1));
i := 0;
while abs((x1-x2)/x2) > 10^(-Digits+1) and i < 10 do
i := i+1 ; x1 := x2; x2 := x1-f(x1)/df(x1) od;
if i = 10 then ‘nope‘ else evalf( x2,Digits-3) fi; end:
newton(x->x^2-2,1.);
```

$$1.414213562$$

Secant Method**:** When the derivative is expensive to compute, one can work with a secant line instead of the tangent line. Briefly, one starts with two starting points x_1, x_2 and computes the x-intercept x_3 of the line through (x_1, f(x_1)) and (x_2, f(x_2)). If $\left|\frac{x_2-x_3}{x_3}\right|$ is too big, update $x_1 = x_2$, $x_2 = x_3$, and repeat.

Problem: Implement the secant method in Maple. Compare it with bisection and Newton, discussing advantages and disadvantages over each.

6.2.1 ***Problem:*** *The function* f$(x) = 1 + x + x^2 - x^3 + x^4 + x^5$ *is 1-1 and decreasing on -1 to -1/2. Write several procedures in Maple to compute the values of the inverse function g on the interval f(-1/2) to f(-1). Export at least one of them to C++.*

1. Using inverse interpolation.
2. Using fsolve to solve equations
3. Using bisect to solve equations
4. Using Newton or secant to solve equations

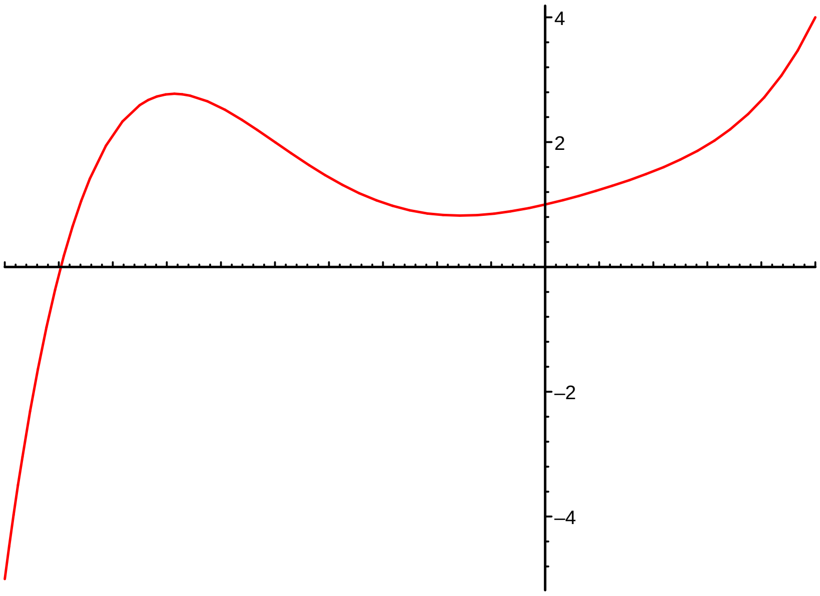
```
restart;
```

Solution: first define the function f and plot it on an inverval containing [-1,-.5] to verify the claim in the problem.

```
f := x-> 1 + x + x^2 - x^3 + x^4 + x^5 ;
```

$$f := x \to 1 + x + x^2 - x^3 + x^4 + x^5$$

```
plot(f,-2..1);
```



So, we can see by inspection that the 1-1 claim could be expanded even out to around -1.5 to -.4. But, we will stick to the original request. We can define the inverse function using fsolve like so.

```
g := proc(y)
local x;
fsolve(f(x)=y,x,-1..-.5) end;
```

$$g := \textbf{proc}(y)\ \textbf{local}\ x;\ \mathrm{fsolve}(\mathrm{f}(x) = y,\ x,\ -1..-.5)\ \textbf{end}$$

```
f(-1);
```

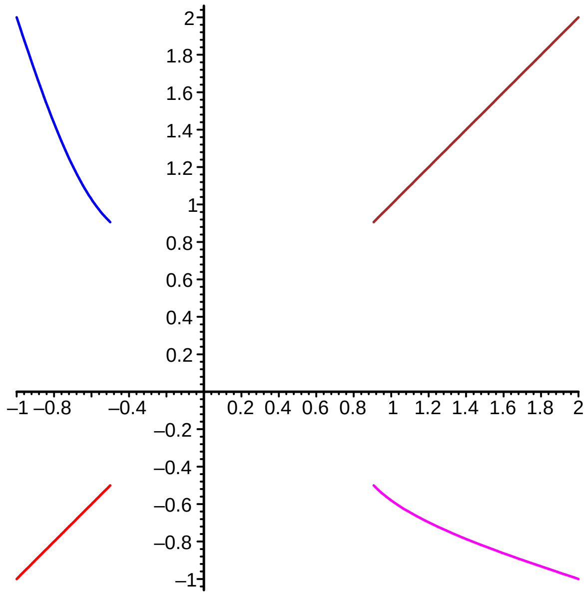
$$2$$

```
g(f(-1/2));
```

$$-.5000000000000000$$

Here is a plot of f restricted to -1..-.5, its inverse g and the compositions fg and gf.

```
plots[display]([plot(f,-1..-.5,color=blue),plot(g,f(-.5)..f(-1),color=magenta),plot(g@f,-
```



6.2.2 ***Exercise:*** *replace the call to Maple's fsolve with a call to bisect in the definition of g above. Compare with the original g.*

Solution: Copy down the code and replace the call . Name the new inverse of f g1.

```
g1 := proc(x)
local y;
bisect(f-x,-1,-.5) end;
```

$$\mathit{g1} := \mathbf{proc}(x)\ \mathbf{local}\ y;\ \mathrm{bisect}(f - x,\ -1,\ -.5)\ \mathbf{end}$$

To compare the functions g and g1 for accuracy and speed use time() to calculate the time of execution.

```
time();
```

$$65.713$$

Here is a small word to compare them.

```
compare := proc(t)
local ti1, ti2,v1,v2;
ti1 := time():  v1 := g(t):  ti1 := time()-ti1;
ti2 := time():  v2 := g1(t):  ti2 := time()-ti2;
[evalf(t),v1,v2,ti1,ti2]; end:
seq(compare(29/32+i*(2-29/32)/10),i=0..10);
```

```
[.9062500000000000, −.5000000000000000, −.5000000000000000, .076, .477],
    [1.015625000000000, −.5917060731930839, −.5917060731930839, .083, .192],
    [1.125000000000000, −.6585691449954331, −.6585691449954331, .069, .467],
```

```
[1.234375000000000, −.7141208746352709, −.7141208746352709, .068, .220],
[1.343750000000000, −.7630262604026249, −.7630262604026249, .068, .207],
[1.453125000000000, −.8075731656468915, −.8075731656468915, .343, .207],
[1.562500000000000, −.8491056144662296, −.8491056144662296, .067, .207],
[1.671875000000000, −.8885158840674250, −.8885158840674250, .068, .467],
[1.781250000000000, −.9264587250565646, −.9264587250565646, .083, .207],
[1.890625000000000, −.9634627274190962, −.9634627274190962, .067, .468],
[2., −1., −1.000000000000000, .068, .166]
```

As we can see, both g and g1 return the same values, while g is 4 to 9 times faster than g1. This is because fsolve is faster than bisect. It might use **Newton's method** after it gets close enough to the solution.

Now let's swap out fsolve for newton in our definition of the inverse of the function f given at the start. Call the new inverse g2. Notice that we take for our starting point for all values -.75

```
g2 := proc(x)
local y;
newton(f-x,-.75) end;
```

```
g2 := proc(x) local y; newton(f − x, −.75) end
```

Add a line to our compare function.

```
compare := proc(t)
local ti1, ti2,v1,v2,ti3,v3;
ti1 := time():  v1 := g(t):  ti1 := time()-ti1;
ti2 := time():  v2 := g1(t):  ti2 := time()-ti2;
ti3 := time():  v3 := g2(t):  ti3 := time()-ti3;
[evalf(t),v1,v2,v3,ti1,ti2,ti3]; end:
seq(compare(29/32+i*(2-29/32)/10),i=0..10);
```

```
[.9062500000000000, −.5000000000000000, −.5000000000000000, .068, .234],
    [1.015625000000000, −.5917060731930839, −.5917060731930839, .068, .467],
    [1.125000000000000, −.6585691449954331, −.6585691449954331, .083, .206],
    [1.234375000000000, −.7141208746352709, −.7141208746352709, .068, .206],
    [1.343750000000000, −.7630262604026249, −.7630262604026249, .344, .206],
    [1.453125000000000, −.8075731656468915, −.8075731656468915, .082, .206],
    [1.562500000000000, −.8491056144662296, −.8491056144662296, .069, .467],
    [1.671875000000000, −.8885158840674250, −.8885158840674250, .069, .206],
    [1.781250000000000, −.9264587250565646, −.9264587250565646, .081, .468],
    [1.890625000000000, −.9634627274190962, −.9634627274190962, .068, .207],
    [2., −1., −1.000000000000000, .068, .165]
```

Upon inspection, we see that Newton returns the same values at about the same speed that fsolve does.

Exercise: Change the definition of g2 so that the starting value is -.5 instead of -.75. Explain the behaviour. Under what conditions on g might any single choice for the starting value not work everywhere in [a,b].

7 Notes on writing C++ programs with the gnu compiler

To do anything but the smallest 'hello world' program, it is good programming style to break your program into several pieces (files) and compile and test them separately before compiling the final program (in reality, there is no final program). This meshes very well with the way we do mathematical research, or solve mathematical problems at any level – we break the problem into smaller problems which we solve separately and then construct the solution to the original problem from the pieces.

7.1 The compiler options -c and -o file

The gnu compiler gcc (or g++) has two options that we use a lot when compiling our program in pieces.

The compile option -c This is used on a single source (.cpp) file to construct an objective file (.o) with the same name. If we have a source file called *bisect.cpp (implementing the bisection method for solving an equation) that we want to compile, the command*

```
g++ -c bisect.cpp
```

will set the compiler off on the task of trying to create a compiled version which it calls *bisect.o. (Note: If you are using the C++ compiler from the Chisel disk, the gnu compiler name is gcc, rather than g++.) Of course, there will probably be errors in the program the first time. But with persistence, you will soon have a bisect.o. Then what? Well, you need a main.cpp to test out bisect.o. This would be compiled separately with the command*

```
g++ -c main.cpp
```

You only compile one .cpp file at a time with the -c option.

Now the second option comes in when you want to compile an executable file:

The compile option -o file : Use this option to link your .o files into an excutable file called *file* . So, to compile an excutable program called *roots* which tests the bisection method, give the command

```
g++ -o roots main.o bisect.o
```

There may be compiler errors here too. If you are using the compiler on the Chisel disk, you need to load the library gpp or there will will be errors. So, the command there reads

```
gcc -o roots main.o bisect.o -lgpp
```

Now if everything went well, we can just say the word *roots* to run the program.

Of course, even if the program runs, there probably will be changes for you to put in. So you will find yourself issuing the same sequence of commands over and over. This is where the program make comes in handy.

7.2 Using make to manage your project

The program make was written to simplify the development of a program that you are working on. Most programmers use it or something like it to develop projects. A simple makefile to work on the project roots would look something like this. (Note the tabs at the beginning of the command lines, these are important).

```
# a makefile for the project roots
roots : main.o bisect.o
<tab> g++ -o roots main.o bisect.o
main.o : main.cpp
<tab> g++ -c main.cpp
bisect.o : bisect.cpp
<tab> g++ -c bisect.cpp
clean :
<tab> rm roots main.o bisect.o
```

This would be put in a file *makefile* in the directory with the source files, and from there the command *make roots* would compile any files that had been changed since the last time the command was issued. If you want to start over, you would issue the command *make clean.*

7.3 Creating libraries of procedures with the program ar

Suppose we have written and tested some other rootfinding methods, say Newton's method (*newton.cpp*) and the secant method (*secant.cpp*). At that point, we might want to start a *library of compiled procedures* with these three. This can be done with the command

```
ar ru mylib.a bisect.o newton.o secant.o
```

ar is a program which archives procedures, ru is an option meaning replace and update, mylib.a is the name we have chosen for our library, and the three .o files have been compiled already.

Problem. There is a zipfile here called roots.zip which contains files to get you started on building the library mylib.a. Download it and unzip it. Finish the task by writing, compiling, and testing newton.o and secant.o. Then add newton.o and secant.o to the library. Document the source .cpp file with a description of the algorithm and its syntax.

8 Debugging in Maple and C++

We have all had ample opportunity to have things go wrong in both Maple and C++ at various times. If you haven't, work harder, and it will happen. Here we will codify some of the debugging techniques and tools that are readily available for our use in these trying times.

Roughly speaking, in any development environment, there are two places where things go wrong: at compile time and at run time. So, if we are trying to export procedures from Maple to C++, there are roughly four places to expect programming errors: At Maple 'compile time', at Maple run time, at C++ compile time, and at C++ run time. Wait a minute, you say, why don't I just skip the Maple and go straight to C++, if my goal is to get C++ code? The answer is, go right ahead, everybody else does that anyway. However, compile time debugging in Maple is much more enjoyable than it's counterpart in C++ and can replace some of the C++ compile time debugging, especially the part dealing with mathematical errors made in the procedure definition.

8.1 Debugging in a Maple worksheet.

Suppose we are trying to develop a function in Maple to compute the nth term of the fibonaci sequence:

$x_1 = 1$, $x_2 = 1$, $x_n = x_{n-1} + x_{n-2}$ for $2 < n$. We can do this iteratively, first in an output cell.

```
n := 10:
x[1]:= 1:  x[2] := 1:
for i from 3 to n do x[i] := x[i-1]+x[i-2]:  od:
x[n];
```

```
55
```

We can define this as a function, by just copying down the relavant input cells to a new input cell, and sticking on a proc line and an end line. (some other modifications may be necessary.) Then when we execute the input cell, the definition will (or will not) 'compile'. If it does, and you have a semicolon after the end, you will see a nicely formatted definition of the word.

```
fibo := proc(n)
local x,i;
x[1]:= 1:  x[2] := 1:
for i from 3 to n do x[i] := x[i-1]+x[i-2]:  od:
x[n];
end;
```

$$
\begin{array}{l}
\mathit{fibo} := \mathbf{proc}(n) \\
\quad \mathbf{local}\, x,\, i; \\
\qquad x_1 := 1\,;\, x_2 := 1\,;\ \ \mathbf{for}\, i\, \mathbf{from}\, 3\, \mathbf{to}\, n\, \mathbf{do}\, x_i := x_{i-1} + x_{i-2}\, \mathbf{od}\,;\, x_n \\
\quad \mathbf{end}
\end{array}
$$

At that point, you can test the code.

```
fibo(3);
```

```
2
```

There are many things that can happen that will make the code fail to compile: Here are a couple of examples.

See if you can find the mistake.

```
fibo := proc(n)
```

```
    local x,i;
    x[1]:= 1 x[2] := 1:
    for i from 3 to n do x[i] := x[i-1]+x[i-2]:  od:
    x[n];
    end;
```

```
missing operator or `;`
```

```
    fibo := proc(n);
    local x,i;
    x[1]:= 1:  x[2] := 1:
    for i from 3 to n do x[i] := x[i-1]+x[i-2]:  od:
    x[n];
    end;
```

```
reserved word `local` unexpected
```

Anyway, after we get something to compile, then we have to run it. This can reveal a whole new set of errors: For example, here is a version of fibo that compile, but still has an error

```
    fibo := proc(n)
    local x,i;
    x[1]:= 1:  x[2] := 1:
    for i from 1 to n do x[i] := x[i-1]+x[i-2]:  od:
    x[n];
    end;
```

```
  fibo := proc(n)
local x, i;
    x[1] := 1;
    x[2] := 1;
    for i to n do x[i] := x[i - 1] + x[i - 2] od;
    x[n]
end
```

So calculate the fifth term in the fibonaci sequence.

```
    fibo(5);
```

```
                                    8 x[0] + 5 x[-1]
```

Clearly, something is wrong. We can probably discover the error by inspecting our definition. (this takes care of many runtime problems.) If that fails, we can set printlevel to 10 (or some higher number) to printout what is being computed when we call fibo.

```
    printlevel:=10;
```

```
                                    printlevel := 10
```

```
    fibo(3);
```

```
{--> enter fibo, args = 3
                                        x[1] := 1
                                        x[2] := 1
                                  x[1] := x[0] + x[-1]
                                 x[2] := 2 x[0] + x[-1]
                                x[3] := 3 x[0] + 2 x[-1]
                                    3 x[0] + 2 x[-1]
<-- exit fibo (now at top level) = 3*x[0]+2*x[-1]}
                                    3 x[0] + 2 x[-1]
```

Now the error is obvious. We set x_1 and x_2 to 1, but then started our loop at $i = 1$, when we should have started at $i = 3$.

```
printlevel:=1;
```

```
printlevel := 1
```

The technique of setting *printlevel* := 10 is the first thing you try after inspection to find run time errors. If you have several procedures being used and you only want to see what is happening in one of them, you can use debug

```
debug(fibo);
```

$$fibo$$

```
fibo(3);
```

```
{--> enter fibo, args = 3
```

$$x_1 := 1$$
$$x_2 := 1$$
$$x_1 := x_0 + x_{-1}$$
$$x_2 := 2\,x_0 + x_{-1}$$
$$x_3 := 3\,x_0 + 2\,x_{-1}$$
$$3\,x_0 + 2\,x_{-1}$$

```
<-- exit fibo (now at top level) = 3*x[0]+2*x[-1]}
```

$$3\,x_0 + 2\,x_{-1}$$

Of course, when you have found the error and don't want to see all that output anymore, you can set printlevel := 1 again or use undebug.

```
undebug(fibo);
```

$$fibo$$

```
fibo(3);
```

$$3\,x_0 + 2\,x_{-1}$$

So let's copy down a working version of fibo

```
fibo := proc(n)
local x,y,z,i;
if n = 1 or n =2 then z := 1 else
x:= 1:  y := 1:
for i from 3 to n do z := x+y; x := y; y := z; od:
fi;
z;
end:
```

We have seen how to time a procedure in Maple using time()

```
ti := time():
fibo(500);
ti1 := time():
`time of execution = `,ti1-ti;
fibo(500);
ti2 := time():
`2nd time of execution = `,ti2-ti1;
```

```
139423224561697880139724382870407283950070256587697307264108962943\
    8325571622863290691557658876222521294125
```

time of execution = , .010

139423224561697880139724382870407283950070256587697307264410896294\
8325571622863290691557658876222521294125
2nd time of execution = , .010

We can also profile a procedure, that is compile statistics on how many times a loop in the procedure is executed, etc. To do it, you first have to read in the profile defnition
.380

```
readlib(profile);
```

proc() . . . **end**

```
profile(fibo);
fibo(500);
```

139423224561697880139724382870407283950070256587697307264410896294\
8325571622863290691557658876222521294125

```
showprofile(fibo);
```

```
function           depth    calls     time    time%         bytes
bytes%
----------------------------------------------------------------------
-----
fibo                   1        3     .030   100.00        150308
99.92
----------------------------------------------------------------------
-----
total:                 2        4     .030   100.00        150424
100.00
```

Timing and profiling are mainly useful techniques which one would use if one had two ways to do the same thing and wanted to compare them. (That's not quite true, you would want to profile a complicated definition to see if there are parts that are not being executed, or where the program is spending most of its time.)

There are other ways to define the fibonaci sequence: Here is a recursive definition.

```
fibo2 := proc(n)
options remember;
if n = 1 or n = 2 then 1 else
fibo2(n-1)+fibo2(n-2) end;
```

```
reserved word ‘end‘ unexpected
```

Oh, oh. A compile time error. What is it? This is a common error, one which I make a lot, so I can usually find it, even though the compiler does not spell out the error.

```
fibo2 := proc(n)
options remember;
if n = 1 or n = 2 then 1 else
fibo2(n-1)+fibo2(n-2) fi end:
```

```
ti := time():
fibo2(500);
ti1 := time():
‘time of execution = ‘,ti1-ti;
fibo2(500);
ti2 := time():
‘2nd time of execution = ‘,ti2-ti1;
```

```
1394232245616978801397243828704072839500702565876973072641089629\4\
    83255716228632906915576588762225212941 25
                        time of execution = , .020
```

Notice that the first time through, the execution time is comparable with fibo, but the second time it is much faster. That's because it just looks up the value, which it has computed before.

```
profile(fibo2);
fibo2(500);
```

```
13942322456169788013972438287040728395007025658769730726410896294\
    832557162286329069155765887622252129412 5
```

```
showprofile(fibo2);
```

```
function           depth    calls     time    time%         bytes
bytes%
----------------------------------------------------------------------
-----
fibo2                  1        1    0.000     0.00           116
.15
----------------------------------------------------------------------
-----
total:                 2        3     .020   100.00         75704
100.00
```

8.2 Debugging exported code with dbg and gprof

Now when we have some code we have exported to C++, we have learned how to get it to run by paying

close attention to the compiler messages. If we want to check run time errors, we need to include the compile option -g in our compile command, so that we can use the gnu debugger gdb.

So, for example, if we wanted to be able to use the gcc debugger gdb on the program fibo.cc we are going to construct below, we would compile it like so

g++ -g -o cmpfib fibo.cc -I../include -lm

Then to run the program under control of gdb, we would run

gdb cmpfib

gdb has very good online help, so that with very little help in getting started you can learn how it works by experiment.

If in addition, we want to profile the execution of the program to see where most of the time is spent etc, we need to include the compile option -pg in the compile command so that we can use the gnu profiler *gprof.*

So, for example, if we wanted to be able to use gprof on the program fibo.cc we are going to construct below, we would compile it like so *g++ -g -o cmpfib fibo.cc -I../include -lm -gp*

Then to get some statistics, run the program cmpfib a couple of times, then run gprof

gprof cmpfib

This command will create much more information than you can use. We can reduce the information by choosing some options (See the man pages on gprof)

gprof -a -b -f fibo -f fibo2 cmpfib

So lets load in codegen and export fibo and fibo2 and compare them in C++

```
with(codegen);
```

$$[C, GRAD, GRADIENT, HESSIAN, JACOBIAN, cost, declare, dontreturn, fortran, horner, intrep2maple, joinprocs, makeglobal, makeparam, makeproc, makevoid, maple2intrep, optimize, packargs, packlocals, packparams, prep2trans, renamevar, split, swapargs]$$

First, write a little main to export too.

```
main := proc()
local n;
n := readstat(`which term?  `);
print(fibo(n), fibo2(n));
end;
```

$$main := \mathbf{proc}()\,\mathbf{local}\,n;\ n := \mathrm{readstat}(`\textit{which term?}\ \ `);\ \mathrm{print}(\mathrm{fibo}(n), \mathrm{fibo2}(n))\ \mathbf{end}$$

```
10;
```

$$55, 55$$

```
main();
20;
```

$$6765, 6765$$

So now lets export to ansi c

```
C(fibo,ansi);
C(fibo2,ansi);
C(main,ansi);
```

```
int fibo(double n)
{
  int x;
  int y;
  int z;
  int i;
  {
    if( n == 1.0 || n == 2.0 )
      z = 1;
    else
      {
        x = 1;
        y = 1;
        for(i = 3;i <= n;i++)
        {
          z = x+y;
          x = y;
```

```
          y = z;
        }
      }
    return(z);
  }
}
/* The options were    : remember */
#include <math.h>
double fibo2(double n)
{
  {
    if( n == 1.0 || n == 2.0 )
      return(1.0);
    else
      return(fibo2(n-1.0)+fibo2(n-2.0));
  }
}
#include <math.h>
void main(void)
{
  double n;
  {
    n = readstat(which term?  );
C/expression:   Unknown function:   fibo   will be left as is.
C/expression:   Unknown function:   fibo2   will be left as is.
    printf( "%e%e\n" ,fibo(n),fibo2(n));
    return;
  }
}
```

After we massage this file and save it to, say cmpfib.cc, it will compile and run.

#include <math.h> #include <iostream.h> // int fibo(int n) { int x; int y; int z; int i; { if(n == 1 —— n == 2) z = 1; else { x = 1; y = 1; for(i = 3;i <= n;i++) { z = x+y; x = y; y = z; } } return(z); } }

double fibo2(int n) { { if(n == 1 —— n == 2) return(1); else return(fibo2(n-1)+fibo2(n-2)); } } // void main(void) { int n; { cout << "which term? "; cin >> n; cout << fibo(n) << " " << fibo2(n); return(0); } }

If we compile with the command g++ -g -o cmpfib cmpfib.cc -I../include -lm -gp

we can then do a runtime debugging session with gbd. Here is a script of one where I ran it twice sucessfully and then gave it some bad input.

```
 Script started on Wed Mar  4 12:36:58 1998
t5% gdb cmpfib
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for
details.
GDB 4.16 (i386-unknown-bsdi3.0),
Copyright 1996 Free Software Foundation, Inc...
(gdb) list
37            return(fibo2(n-1)+fibo2(n-2));
38        }
39      }
40
```

```
41
42      void main(void)
43      {
44        int n;
45        {
46          cout << ”which term?  ”;
(gdb) run
Starting program:
/a/u4/u/disk2/ma/carl/public_html/ma777/mvmap/mon32/cmpfib
which term?  5
5  5
Program exited normally.
(gdb) run
Starting program:
/a/u4/u/disk2/ma/carl/public_html/ma777/mvmap/mon32/cmpfib
which term?  10
55  55
Program exited normally.
(gdb) run
Starting program:
/a/u4/u/disk2/ma/carl/public_html/ma777/mvmap/mon32/cmpfib
which term?  t
Program received signal SIGSEGV, Segmentation fault.
0x112c in fibo2 (n=-104804) at cmpfib.cc:37
37            return(fibo2(n-1)+fibo2(n-2));
(gdb) help
List of classes of commands:
running – Running the program
stack – Examining the stack
data – Examining data
breakpoints – Making program stop at certain points
files – Specifying and examining files
status – Status inquiries
support – Support facilities
user-defined – User-defined commands
aliases – Aliases of other commands
obscure – Obscure features
internals – Maintenance commands
Type ”help” followed by a class name for a list of commands in that
class.
Type ”help” followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) help data stack
Examining the stack.
The stack is made up of stack frames.  Gdb assigns numbers to stack
```

```
frames
counting from zero for the innermost (currently executing) frame.
At any time gdb identifies one frame as the ”selected” frame.
Variable lookups are done with respect to the selected frame.
When the program being debugged stops, gdb selects the innermost
frame.
The commands below can be used to select other frames by number or
address.
List of commands:
bt – Print backtrace of all stack frames
backtrace – Print backtrace of all stack frames
select-frame – Select a stack frame without printing anything
frame – Select and print a stack frame
down – Select and print stack frame called by this one
up – Select and print stack frame that called this one
return – Make selected stack frame return to its caller
Type ”help” followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb) bt
#0  0x112c in fibo2 (n=-104804) at cmpfib.cc:37
#1  0x1132 in fibo2 (n=-104803) at cmpfib.cc:37
#2  0x1132 in fibo2 (n=-104802) at cmpfib.cc:37
#3  0x1132 in fibo2 (n=-104801) at cmpfib.cc:37
#4  0x1132 in fibo2 (n=-104800) at cmpfib.cc:37
#5  0x1132 in fibo2 (n=-104799) at cmpfib.cc:37
#6  0x1132 in fibo2 (n=-104798) at cmpfib.cc:37
#7  0x1132 in fibo2 (n=-104797) at cmpfib.cc:37
#8  0x1132 in fibo2 (n=-104796) at cmpfib.cc:37
#9  0x1132 in fibo2 (n=-104795) at cmpfib.cc:37
#10 0x1132 in fibo2 (n=-104794) at cmpfib.cc:37
—Type <return> to continue, or q
<return> to quit—qQuit
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y
(˜/public_html/ma777/mvmap/mon32)t5%
Use ”exit” to leave sh.
(˜/public_html/ma777/mvmap/mon32)t5% exit
exit
Script done on Wed Mar  4 12:39:28 1998
```

What we can see is what putting in crappy data (the letter t) does to the program (puts it into an infinite loop)

We can also profile the statistics with gprof

```
gprof -a -b -f fibo -f fibo2 cmpfib
```

None granularity: each sample hit covers 4 byte(s) for 33.33% of 0.03 seconds

```
                              called/total       parents
index  %time    self descendents  called+self    name    index
                              called/total       children
granularity: each sample hit covers 4 byte(s) for 33.33% of 0.03
seconds
 %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.3       0.01      0.01        7     1.43     1.43
__thread_aio_poll (5)
 33.3       0.02      0.01        6     1.67     1.96  ___dtoa (2)
 33.3       0.03      0.01
__IO_do_write (4)
  0.0       0.03      0.00       25     0.00     0.00
__thread_sys_write (29)
  0.0       0.03      0.00       25     0.00     0.00  _sigprocmask
(30)
  0.0       0.03      0.00        9     0.00     0.00  _ioctl (31)
  0.0       0.03      0.00        8     0.00     1.29  ___svfprintf
<cycle 2> (3)
  0.0       0.03      0.00        7     0.00     0.00  _malloc (32)
  0.0       0.03      0.00        7     0.00     0.56  _memchr
<cycle 2> (9)
  0.0       0.03      0.00        6     0.00     0.00
__thread_timer_dotimeouts (33)
  0.0       0.03      0.00        5     0.00     0.57  _memmove (11)
  0.0       0.03      0.00        4     0.00     0.00  ___ad_d2b
(34)
  0.0       0.03      0.00        4     0.00     0.18  ___qdivrem
(20)
  0.0       0.03      0.00        3     0.00     0.00  ___sflffree
(35)
  0.0       0.03      0.00        3     0.00     1.43  ___sflfinit
(7)
  0.0       0.03      0.00        3     0.00     0.00  ___swsetup
(36)
  0.0       0.03      0.00        3     0.00     0.19
__thread_aio_suspend (21)
  0.0       0.03      0.00        3     0.00     0.00
```

```
__thread_sys_open (37)
  0.0     0.03    0.00      3    0.00    1.43  _abort (8)
  0.0     0.03    0.00      3    0.00    0.00  _fibo2__Fi
(38)
  0.0     0.03    0.00      3    0.00    0.00
_pthread_mutex_destroy (39)
  0.0     0.03    0.00      2    0.00    0.00  ___fixdfsi
(40)
  0.0     0.03    0.00      2    0.00    0.00  ___swhatbuf
(41)
  0.0     0.03    0.00      2    0.00    0.00  _close (42)
  0.0     0.03    0.00      2    0.00    0.71
_pthread_mutex_lock (15)
  0.0     0.03    0.00      2    0.00    0.00  _write (43)
  0.0     0.03    0.00      1    0.00    0.57  ___srefill
(22)
  0.0     0.03    0.00      1    0.00    0.00  ___sysctl
(44)
  0.0     0.03    0.00      1    0.00    0.00
__thread_queue_deq (45)
  0.0     0.03    0.00      1    0.00    1.43
__thread_queue_enq (16)
  0.0     0.03    0.00      1    0.00    0.19  _fibo__Fi
<cycle 1> (26)
  0.0     0.03    0.00      1    0.00    0.00  _lseek (46)
  0.0     0.03    0.00      1    0.00    2.86  _main
<cycle 1> (12)
  0.0     0.03    0.00      1    0.00    0.00  _snprintf (47)
  0.0     0.03    0.00      1    0.00    2.86  _strtol (13)
```

Index by function name

```
   (4) __IO_do_write        (21) __thread_aio_suspen  (32)
_malloc
  (34) ___ad_d2b            (45) __thread_queue_deq    (9)
_memchr
   (2) ___dtoa              (16) __thread_queue_enq   (11)
_memmove
  (40) ___fixdfsi           (37) __thread_sys_open    (39)
_pthread_mutex_dest
  (20) ___qdivrem           (29) __thread_sys_write   (15)
_pthread_mutex_lock
  (35) ___sflffree          (33) __thread_timer_doti  (30)
_sigprocmask
   (7) ___sflfinit           (8) _abort               (47)
_snprintf
  (22) ___srefill           (42) _close               (13)
```

```
_strtol
   (3) ___svfprintf          (38) _fibo2__Fi            (43)
_write
  (41) ___swhatbuf           (26) _fibo__Fi             (10)
<cycle 1>
  (36) ___swsetup            (31) _ioctl                 (1)
<cycle 2>
  (44) ___sysctl             (46) _lseek
   (5) __thread_aio_poll     (12) _main
```

9 Building a Maple to C++ library

The Maple language has a large builtin vocabulary of problem solving words. It would be nice to have a library of C++ procedures whose names and syntax correspond with Maple words and syntax. We propose to start the library here. As the library grows, it will become easier to export Maple code to C++ code.

To begin, we will concentrate on building a library of procedures like those in *linalg*, the package of linear algebra words in Maple. Here are the words.

```
with(linalg);
```

```
Warning, new definition for norm
Warning, new definition for trace
  [BlockDiagonal, GramSchmidt, JordanBlock, LUdecomp, QRdecomp,
Wronskian,

        addcol, addrow, adj, adjoint, angle, augment, backsub, band,
basis,

        bezout, blockmatrix, charmat, charpoly, cholesky, col, coldim,
colspace,

        colspan, companion, concat, cond, copyinto, crossprod, curl,
definite,

        delcols, delrows, det, diag, diverge, dotprod, eigenvals,
eigenvalues,

        eigenvectors, eigenvects, entermatrix, equal, exponential,
extend,

        ffgausselim, fibonacci, forwardsub, frobenius, gausselim,
gaussjord,

        geneqns, genmatrix, grad, hadamard, hermite, hessian, hilbert,

        htranspose, ihermite, indexfunc, innerprod, intbasis, inverse,
ismith,

        issimilar, iszero, jacobian, jordan, kernel, laplacian,
leastsqrs,

        linsolve, matadd, matrix, minor, minpoly, mulcol, mulrow,
multiply, norm,

        normalize, nullspace, orthog, permanent, pivot, potential,
randmatrix,

        randvector, rank, ratform, row, rowdim, rowspace, rowspan,
rref,

        scalarmul, singularvals, smith, stackmatrix, submatrix,
subvector,

        sumbasis, swapcol, swaprow, sylvester, toeplitz, trace,
transpose,

        vandermonde, vecpotent, vectdim, vector, wronskian]
```

So let's do *addrow* and *addcol*. I have copied the descriptions of these words from the help sheet on addrow (it is the same sheet as addcol) into this worksheet.

```
?addrow
```

9.1 linalg[addrow] - form linear combinations of matrix rows

linalg[addcol] - form linear combinations of matrix columns
Calling Sequence:
addrow(A, r1, r2, m)
addcol(A, c1, c2, m)
Parameters:

A - matrix
r1, r2, c1, c2 - integers; row (column) indices
m - a scalar expression

Description:

E

The call addrow(A, r1, r2, m) returns a copy of the matrix A in which row r2 is replaced by m*row(A,r1) + row(A,r2). Similarly addcol(A, c1, c2, m) returns a copy of the matrix A in which column c2 is replaced by m*col(A,c1) + col(A,c2).

These functions are part of the linalg package, and so can be used in the form addrow(..) only after performing the command with(linalg) or with(linalg,addrow). These functions can always be accessed in the long form linalg[addrow](..).

Notice that *addrow* and *addcol* return a modified copy of A. So, if you wanted to update the matrix A with addrow or addcol, you would have to issue a command like

A := addrow(A,r1,r2,m);

To illustrate,

```
A := linalg[randmatrix](3,3);
```

$$A := \begin{bmatrix} 43 & -62 & 77 \\ 66 & 54 & -5 \\ 99 & -61 & -50 \end{bmatrix}$$

```
addrow(A,2,3,10), evalm(A);
```

$$\begin{bmatrix} 43 & -62 & 77 \\ 66 & 54 & -5 \\ 759 & 479 & -100 \end{bmatrix}, \begin{bmatrix} 43 & -62 & 77 \\ 66 & 54 & -5 \\ 99 & -61 & -50 \end{bmatrix}$$

```
A := addrow(A,2,3,10), evalm(A);
```

$$A := \begin{bmatrix} 43 & -62 & 77 \\ 66 & 54 & -5 \\ 759 & 479 & -100 \end{bmatrix}, \begin{bmatrix} 43 & -62 & 77 \\ 66 & 54 & -5 \\ 99 & -61 & -50 \end{bmatrix}$$

A more efficient way to do this would be to have addrow update A itself in the procedure. This would avoid making an unnecessary copy of A. We can accomplish this in C++ by passing A by reference, rather than value. But first we want to export addrow.

Our method for doing this will be to define an addrow in Maple that fits the description in the help sheet, and then exporting that using the C word from the package *codegen* (in release 5). Note: We could try to export the actual definition, but usually that has so much extra stuff that it is not worth doing. Besides, we want to develop the code in Maple ourselves.

So here is a definiton of addrow with the same syntax as Maples addrow that works in Maple.

```
addrow := proc(A,r1,r2,m)
local c,i;
c := rowdim(A):
for i from 1 to c do
A[r2,i]:= A[r2,i]+m*A[r1,i]:
od:
evalm(A);
end:
```

We should test this to make sure it works.

```
A := linalg[randmatrix](3,3);
```

$$A := \begin{bmatrix} -61 & 41 & -58 \\ -90 & 53 & -1 \\ 94 & 83 & -86 \end{bmatrix}$$

```
addrow(A,2,3,10);
```

$$\begin{bmatrix} -61 & 41 & -58 \\ -90 & 53 & -1 \\ -806 & 613 & -96 \end{bmatrix}$$

ok, it works. Now export it to ansi C

```
with(codegen):
```

```
Warning, new definition for C
```

```
C(addrow,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
A[r2,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[r2,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[r1,i]
#include <math.h>
double addrow(double A,int r1,int r2,double m)
{
  double c;
  int i;
  {
C/expression:   Unknown function:   rowdim   will be left as is.
    c = rowdim(A);
    for(i = 1;i <= c;i++)
      A[r2-1][i-1] += m*A[r1-1][i-1];
C/expression:   Unknown function:   evalm   will be left as is.
    return(evalm(A));
```

```
  }
}
```

We can take our mouse cursor, color the exported text and copy it into a text file using vi say and then modify that to get a C++ version of addrow that works. I have copied back into this worksheet the working versions of addrow and addcol we obtained starting from the code above. Below that I have copied a main to test them. Below that I have copied a makefile (one that works with g++ on the t-machines, and one that works with the gcc on the Chisel disk) that compiles, tests, and archives addrow.o et al into the beginnings of a library of C++ procedures to facilitate developing numerical algorithms in Maple and exporting them to C.

```
//
//  this addrow has the same syntax as linalg[addrow] in Maple
//  It returns a copy of A which has been modified (A is left alone)
//  Note:  A is passed in by reference (note &) to save making a copy
//  but is made constant (note const) to avoid accidental
modification.
//
#include <math.h>
#include "vecdefs.h"
#include "mvm.h"
MATRIX_double addrow(const MATRIX_double &A,int
r1,int r2,double m)
{
  int c;
  int i;
  {
    MATRIX_double Acpy = A;
    c = A.dim(1);
    for(i = 1;i <= c;i++)
      Acpy(r2-1,i-1) += m*Acpy(r1-1,i-1);
     return(Acpy);
  }
}
//  Here is an addrow which modifies the original matrix A,
//  which is passed by reference (note the & and also the
//  void in front of addrowip)
//  I called it addrowip  (ip = inplace)
//
void addrowip(MATRIX_double &A,int r1,int r2,double
m)
{
  int c;
  int i;
  {
    c = A.dim(1);
    for(i = 1;i <= c;i++)
```

```
      A(r2-1,i-1) += m*A(r1-1,i-1);
  }
}
// We can modify addrow(ip)  to get addcol(ip)
//
MATRIX_double addcol(const MATRIX_double &A,int
c1,int c2,double m)
{
  int r;
  int i;
  {
    MATRIX_double Acpy = A;
    r = A.dim(0);
    for(i = 1;i <= r;i++)
      Acpy(i-1,c2-1) += m*Acpy(i-1,c1-1);
    return(Acpy);
  }
}
// And then modify addcol to get addcolip
//
void addcolip(MATRIX_double &A,int c1,int c2,double
m)
{
  int r;
  int i;
  {
    r = A.dim(0);
    for(i = 1;i <= r;i++)
      A(i-1,c2-1) += m*A(i-1,c1-1);
  }
}
//  Here is a main to test addrow and it variants.
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<iostream.h>
#include "vecdefs.h"
#include <mvm.h>
//
//
MATRIX_double addrow(const MATRIX_double &, int, int, double);
void addrowip(MATRIX_double &, int, int, double);
MATRIX_double addcol(const MATRIX_double &, int, int, double);
void addcolip(MATRIX_double &, int, int, double);
//
```

```
int m;
void main(void)
{
 m=4;
//
 MATRIX_double W(m,m,1.0);
//
 MATRIX_double X(m,m,0);
cout << W << X
<< endl;
//
//
addrowip(W,2,3,10.0);
   cout << ”addrow
inplace\n” << W
<< endl;
//
X=addrow(W,2,3,10.0);
   cout << ”addrow\n”
<< X << endl;
//
addcolip(W,2,3,10.0);
   cout << ”addcol
inplace\n” << W
<< endl;
//
X=addcol(W,2,3,10.0);
   cout << ”addcol\n”
<<X << endl;
//
}

#  Makefile for Maple to C++ development library
#
error:
@echo ”+———————————————— ——————————-+”
@echo ”—                    MA  777 Spring 1998
@echo ”—                  Maple to C++  Library                  —”
@echo ”—                                                  —”
@echo ”—  First uncomment the appropriate definition       —”
@echo ”—  of  COMP and LIBS                                  —”
@echo ”—                                                  —”
@echo ”+———————————————————————————————+”
# Uncomment here if you are using gcc from the Chisel disk
 INCLUDE = -I ../include
```

```
 LIBS = ../lib/libmv.a -lm -lgpp
 COMP = gcc
#
# Uncomment here if you are using g++ on a t-machine
# INCLUDE = -I ../include
# LIBS = ../lib/libmv.a -lm
# COMP = g++
#
#
SRCS = elim.cpp mmult.cpp lufact.cpp test.cpp\
       cubsp2.cpp csplfun.cpp mcub2.cpp \
       addrc.cpp testit.cpp
#
TOBJS = test.o\
        mcub2.o\
       testit.o
#
LIBOBS = addrc.o  cubsp2.o csplfun.o elim.o mmult.o lufact.o
#
EXECS =  cubic linalg  maple
#
maple:  testit.o addrc.o
$(COMP) -o maple testit.o addrc.o  $(LIBS)
#
testit.o: testit.cpp
$(COMP) -c $(INCLUDE) testit.cpp
#
addrc.o: addrc.cpp
$(COMP) -c $(INCLUDE) addrc.cpp
 #
linalg:  test.o  elim.o  mmult.o lufact.o
$(COMP) -o linalg test.o elim.o mmult.o lufact.o  $(LIBS)
#
#
#
elim.o :  elim.cpp
$(COMP)  -c   $(INCLUDE)  elim.cpp
#
mmult.o : mmult.cpp
$(COMP) -c  $(INCLUDE) mmult.cpp
#
lufact.o : lufact.cpp
$(COMP) -c  $(INCLUDE) lufact.cpp
#
test.o :  test.cpp
```

```
$(COMP)  -c test.cpp  $(INCLUDE)
#
cubic :   mcub2.o cubsp2.o csplfun.o
$(COMP) -o cubic mcub2.o cubsp2.o csplfun.o $(LIBS)
#
mcub2.o : mcub2.cpp
$(COMP) -c $(INCLUDE) mcub2.cpp
#
cubsp2.o : cubsp2.cpp
$(COMP) -c $(INCLUDE) cubsp2.cpp
#
csplfun.o : csplfun.cpp
$(COMP) -c $(INCLUDE) csplfun.cpp
#
maplib:  $(LIBOBS)
ar rus ../lib/maplib.a $(LIBOBS)
clean :
rm  $(EXECS)  $(TOBJS) $(LIBOBS)
```

10 Additions to maplib.a and a way of using Maple to generate test data for comparison.

Here are three additions to our maple to c++ library: *dotprod*, *norm*, and *angle*. Our implementation of norm is not complete; norm in Maple means much more than our defintion. I have included a copy of the procedure used to test the added procedures. It employs a more flexible method for testing than the previous test procedures.

10.1 linalg[dotprod] - Vector dot (scalar) produc

Calling Sequence:

dotprod(u, v)

Parameters:

u, v - lists of the same length or vectors of the same dimension

Description:

If u and v are real vectors of the same dimension, then dotprod(u,v) is the sum of the pairwise products $u_i\,v_i$.

A Maple implementation

```
dotprod := proc(u,v)
local n,i,s;
if nops(u)<>nops(v) then ERROR(`dimensions don't match`)fi;
n := nops(u);
s := 0:
for i from 1 to n do
s := s + u[i]*v[i] od;
s end:
```

10.2 linalg[norm] - norm of a vector

Calling Sequence:

norm(A)

Parameters:

A vector

Description:

The function norm(A, normname) computes the specified matrix or vector norm for the matrix or vector A.

NOTE: We have to use vnorm instead of norm, because norm is protected. We will change it in the exported code

```
vnorm := proc(v)
sqrt(evalf(dotprod(v,v)))
end;
```

$$vnorm := \mathbf{proc}(v)\,\mathrm{sqrt}(\mathrm{evalf}(\mathrm{dotprod}(v,\,v)))\;\mathbf{end}$$

10.3 linalg[angle] - compute the angle between vectors

Calling Sequence:

angle(u, v)

Parameters:

u, v - n-dimensional vectors

Description:

The call angle(u, v) computes the angle theta between the n-dimensional vectors u and v via the formula cos(theta) = u.v / (—u— —v—), using the Euclidean inner product and norm.

```
angle := proc(u,v)
arccos(dotprod(u,v)/(vnorm(u)*vnorm(v))) end:
```

10.4 Conversion to C++

The three procedures defined above are translated to ansi C and then copied into a text file. Here is the spell for translating into ansi C.

```
with(codegen):
C(dotprod,ansi);
C(vnorm,ansi);
C(angle,ansi);
```

Then the code is massaged by adding some include directives, and changing the declarations of some variables. Especially, the vectors are declared as instances of Pozo's class *VECTOR_double*, which we are using. Here is the modified code, which compiles under g++ by chanting

```
g++ -c mvmap3.cpp -I..\include
```

if you are in the directory mvmap\progs and you have saved the modified code below in a file called mvmap3.cpp

```
 //
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<iostream.h>
#include ”vecdefs.h”
#include ”mvvd.h”
//
double dotprod(const VECTOR_double & u,const
VECTOR_double & v)
{
  double s;
  {
    if( u.size() != v.size() )
      {
        fprintf(stderr, ”dimensions don’t match” );
        exit(1);
      }
```

```
    int  n = u.size();
      s = 0.0;
      for(int i = 1;i <= n;i++)
        s += u[i-1]*v[i-1];
      return(s);
    }
  }
  double norm(const VECTOR_double &v)
  {
    {
      return(sqrt(0.1E1*dotprod(v,v)));
    }
  }
  double angle(const VECTOR_double & u,const
  VECTOR_double & v)
  {
    {
      return(acos(dotprod(u,v)/norm(u)/norm(v)));
    }
  }
```

10.5 Testing the procedures

As mentioned at the begining, our test procedure for these three procedures is more flexible than our previous procedures.

Rather than the test procedure supplying the data, it provides for inputting the data using the c++ word *cin.* That way you can either type the vectors in when you run the test program or (more efficiently) you can read the vectors in from a file using *input redirection(<)*, and write the output from the test procedure using *output redirection* (> or >>). So, For example, we can define some data in a Maple worksheet, use our Maple vocabulary to generate output from the data, then write the data to a file, direct that data to our C++ test procedures and append the output to the file containing the Maple output for later comparison.

```
//test procedure for dotprod, norm, and angle.
//takes input from the terminal or a file (by indirection with <)
//wants a vector dimension n, and two vectors u and v of that dimension
#include <math.h>
#include <stdlib.h>
#include<iostream.h>
#include "vecdefs.h"
#include "mvvd.h"
//
double dotprod(const VECTOR_double & u,const VECTOR_double & v);
double norm(const VECTOR_double &v);
double angle(const VECTOR_double u,const VECTOR_double v);
//
```

```
main() {
int n = 0;
 cin >> n;
VECTOR_double a(n,0);
 for(int i=0;i<n;i++)
 cin >> a(i);
cout << ”u = \n”
<< a ;
VECTOR_double b(n,0);
 for(int i=0;i<n;i++)
 cin >> b(i);
cout << ”v = \n”
<< b;
cout << ”u dot v = \n”
<< dotprod(a,b) <<
endl;
cout << ”norm u = \n”
<< norm(a) <<
endl;
cout << ”angle between u and v =
\n” << angle(a,b)
<< endl;
}
```

We can compile this with the spell

```
g++ -c tmvmap3.cpp -I../include
```

if we are in the same place as above. Then make an executable , say dotprod, with

```
g++ -o dotprod ../lib/libmv.a -lm
```

Let’s generate two vectors u and v of dimension 4 in Maple.

```
u := [3.2,5.4,2.1,5.44]:
v := [0.3222,.222,4.332,7.4321]:
dp := dotprod(u,v);
                    dp := 51.757664
nrm := vnorm(u);
                    nrm := 8.567590093
ang := angle(u,v);
                    ang := .7932580971
```

Now write the vectors u and v to a datafile, say tmvmap3.dat. Note, you will have to change the path here.

```
fd :=
fopen(‘c:\\classes\\ma777\\mvmap\\progs
\\tmvmap3.dat‘,WRITE,TEXT);
                    fd := 2
```

```
writedata(fd,[4,op(u),op(v)]);
fclose(fd);
```

Now if you have the executable dotprod compiled, you can take the input from the file tmvmap3.dat by the command

```
dotprod < tmvmap3.dat
```

The output can be read from the terminal or captured in a file with >. The output I got was

```
u =
3.2
5.4
2.1
5.44
v =
0.3222
0.222
4.332
7.4321
u dot v =
51.7577
norm u =
8.56759
angle between u and v =
0.793258
```

which compare nicely with Maple's computation.

11 More linalg words

11.1 linalg[stackmatrix] - join two or more matrices together vertically

Calling Sequence:

stackmatrix(A, B,...)

Parameters:

A,B,... - matrices or vectors

Description:

The function stackmatrix joins two or more matrices or vectors together vertically, where a vector is interpreted as a row vector. The matrices and vectors must have the same number of columns.

The command with(linalg,stackmatrix) allows the use of the abbreviated form of this command.

```
stackmatrix := proc(A,B)
local C,ra,ca,rb,i,j;
ra := rowdim(A);
ca := coldim(A);
rb := rowdim(B);
C := array(1..ra+rb,1..ca);
for i from 1 to rb do
for j from 1 to ca do
C[i,j]:= B[i,j]:  od od;
for i from rb+1 to rb+ra do
for j from 1 to ca do
C[i,j] := A[i-rb,j] od od:
evalm(C) end:
```

```
u[1,1];
```

$$-4$$

```
u := randmatrix(2,3); v :=
randmatrix(3,3);
```

$$u := \begin{bmatrix} 6 & 72 & -28 \\ -61 & -59 & 6 \end{bmatrix}$$

$$v := \begin{bmatrix} -87 & 72 & -46 \\ -68 & -42 & -47 \\ -32 & 37 & -93 \end{bmatrix}$$

```
b:=stackmatrix(u,v);
```

$$b := \begin{bmatrix} -87 & 72 & -46 \\ -68 & -42 & -47 \\ -32 & 37 & -93 \\ 6 & 72 & -28 \\ -61 & -59 & 6 \end{bmatrix}$$

```
with(codegen);
```

```
Warning, new definition for fortran
```

*[C, GRAD, GRADIENT, HESSIAN, JACOBIAN, cost, declare, dontreturn, fortran,
horner, intrep2maple, joinprocs, makeglobal, makeparam, makeproc, makevoid,
maple2intrep, optimize, packargs, packlocals, packparams, prep2trans,
renamevar, split, swapargs]*

```
C(stackmatrix,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
B[i,j]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[i-rb,j]
#include <math.h>
#include <stdlib.h>
double *stackmatrix(double A,double B)
{
  double *C;
  int ra;
  int ca;
  int rb;
  int i;
  int j;
  {
    ra = rowdim(A);
    ca = coldim(A);
    rb = rowdim(B);
    C = (double *) malloc(sizeof(double)*((ra+rb)*ca));
    for(i = 1;i <= rb;i++)
      for(j = 1;j <= ca;j++)
        C[(i-1)*ca+j-1] = B[i-1][j-1];
    for(i = rb+1;i <= ra+rb;i++)
      for(j = 1;j <= ca;j++)
        C[(i-1)*ca+j-1] = A[i-rb-1][j-1];
    return(C);
  }
}
```

Here is the modified code.

```
#include <math.h>
#include <stdlib.h>
#include ”vecdefs.h”
#include ”mvm.h”
//
MATRIX_double evalm(const MATRIX_double & );
int rowdim(const MATRIX_double & );
int coldim(const MATRIX_double & );
int nops(const VECTOR_double & );
//
MATRIX_double augment(const MATRIX_double &A,const
MATRIX_double & B)
{
  int ra;
  int ca;
  int cb;
  int i;
```

```
  int j;
  {
    ra = rowdim(A);
    ca = coldim(A);
    cb = coldim(B);
    MATRIX_double C(ra,ca+cb);
    for(i = 1;i <= ra;i++)
    {
      for(j = 1;j <= ca;j++)
        C(i-1,j-1) = A(i-1,j-1);;
      for(j = ca+1;j <= ca+cb;j++)
        C(i-1,j-1) = B(i-1,j-ca-1);
    }
    return(evalm(C));
  }
}
```

```
#include <math.h>
#include <stdlib.h>
#include<iostream.h>
#include ”vecdefs.h”
#include ”mvm.h”
//
MATRIX_double stackmatrix(const MATRIX_double
&A,const MATRIX_double &B)
{
  int ra = A.dim(0);
  int ca= A.dim(1);
  int rb= B.dim(0);
  MATRIX_double C(ra+rb,ca);
  int i;
  int j;
  {
    for(i = 1;i <= rb;i++)
      for(j = 1;j <= ca;j++)
        C(i-1, j-1) = B(i-1,j-1);
    for(i = rb+1;i <= ra+rb;i++)
      for(j = 1;j <= ca;j++)
        C(i-1, j-1) = A(i-rb-1,j-1);
    return(C);
  }
}
```

11.2 linalg[augment] - join two or more matrices together horizontally

linalg[concat] - a synonym for augment

Calling Sequence:

augment(A, B, ...)
concat(A, B, ...)

Parameters:

A,B, ... - matrices or vectors

Description:

The function augment joins two or more matrices or vectors together horizontally. A vector is interpreted as a column vector. The matrices and vectors must have the same number of rows.

The command with(linalg,augment) allows the use of the abbreviated form of this command.

```
augment := proc(A,B)
local C,ra,ca,cb,i,j;
ra := rowdim(A);
ca := coldim(A);
cb := coldim(B);
C := array(1..ra,1..ca+cb);
for i from 1 to ra do
for j from 1 to ca do
C[i,j]:= A[i,j]:  od;
for j from ca+1 to ca+cb do
C[i,j]:=B[i,j-ca] od;
od;
evalm(C) end:
```

```
c
:=augment(randmatrix(3,4),randmatrix(3,2));
```

$$c := \begin{bmatrix} -98 & -36 & 40 & 22 & -55 & 25 \\ 5 & -88 & -43 & -73 & 9 & 40 \\ 25 & 4 & -59 & 62 & 61 & 40 \end{bmatrix}$$

```
b :=
augment(randmatrix(3,4),randmatrix(3,2));
```

$$b := \begin{bmatrix} -78 & 62 & 11 & 88 & 57 & -82 \\ 1 & 30 & 81 & -5 & -48 & -11 \\ -28 & 4 & -11 & 10 & 38 & -7 \end{bmatrix}$$

```
with(codegen):
```

```
Warning, new definition for fortran
```

Here are some utility words to keep from having to modify rowdim, coldim, nops, and evalm

```
#include <math.h>
#include <stdlib.h>
#include<iostream.h>
#include "vecdefs.h"
#include "mvm.h"
```

```
MATRIX_double evalm(const MATRIX_double &A)
{
return(A);
}
//
int rowdim(const MATRIX_double &A)
{
return(A.dim(0));
}
int coldim(const MATRIX_double &A)
{
return(A.dim(1));
}
int nops(const VECTOR_double &v)
{
return(v.size());
}
```

```
   C(augment,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
A[i,j]
C/pretreatment:   WARNING: Unable to determine array bounds in
B[i,j-ca]
#include <math.h>
#include <stdlib.h>
double augment(double A,double B)
{
  double *C;
  int ra;
  int ca;
  int cb;
  int i;
  int j;
  {
C/expression:   Unknown function:   rowdim   will be left as is.
    ra = rowdim(A);
C/expression:   Unknown function:   coldim   will be left as is.
    ca = coldim(A);
    cb = coldim(B);
    C = (double *) malloc(sizeof(double)*(ra*(ca+cb)));
    for(i = 1;i <= ra;i++)
    {
      for(j = 1;j <= ca;j++)
        C[(i-1)*(ca+cb)+j-1] = A[i-1][j-1];
      for(j = ca+1;j <= ca+cb;j++)
        C[(i-1)*(ca+cb)+j-1] = B[i-1][j-ca-1];
    }
C/expression:   Unknown function:   evalm   will be left as is.
    return(evalm(C));
  }
}
```

```
#include <math.h> #include <stdlib.h> #include "vecdefs.h" #include "mvm.h"
   MATRIX_double evalm(const MATRIX_double & ); int rowdim(const MATRIX_double & ); int coldim(const
MATRIX_double & ); int nops(const VECTOR_double & );
   MATRIX_double augment(const MATRIX_double &A,const MATRIX_double & B) {
   int ra; int ca; int cb; int i; int j; { ra = rowdim(A); ca = coldim(A); cb = coldim(B);
MATRIX_double C(ra,ca+cb); for(i = 1;i <= ra;i++) { for(j = 1;j <= ca;j++) C(i-1,j-1) = A(i-1,j-1);;
for(j = ca+1;j <= ca+cb;j++) C(i-1,j-1) = B(i-1,j-ca-1); }
   return(evalm(C)); } }
```

11.3 linalg[submatrix] - extract a specified submatrix from a matrix

Calling Sequence:

submatrix(A, Rrange, Crange)
submatrix(A, Rlist, Clist)

Parameters:

A - a matrix
Rrange, Crange - integer ranges of rows/columns of A
Rlist, Clist - lists of integer row/column indices

Description:

The call submatrix(A, Rrange, Crange) returns the submatrix of A selected by the row range Rrange and the column range Crange.

The call submatrix(A, Rlist, Clist) returns the matrix whose (i,j)th element is A[Rlist[i], Clist[j]].

The command with(linalg,submatrix) allows the use of the abbreviated form of this command.

```
submatrix := proc(A,r,c)
local i,j,n,m,B;
m := r[2]+1-r[1]:
n := c[2]+1-c[1]:
B := array(1..m,1..n);
for i from 1 to m do
for j from 1 to n do
B[i,j] := A[r[1]-1 +i,c[1]-1+j] od od;
evalm(B)
end:
B := linalg[randmatrix](6,7) ;
```

$$B := \begin{bmatrix} 46 & -20 & 52 & 31 & 65 & 46 & -24 \\ 63 & 73 & 76 & -68 & 67 & -38 & 23 \\ 14 & 58 & -92 & -8 & 36 & -90 & 96 \\ -19 & -4 & 38 & -61 & -58 & -91 & 45 \\ -79 & -27 & 32 & -24 & -46 & 12 & 81 \\ 63 & -85 & -36 & -35 & 11 & 90 & 31 \end{bmatrix}$$

```
print(submatrix(B ,[2,4],[3,6]));
```

$$\begin{bmatrix} 76 & -68 & 67 & -38 \\ -92 & -8 & 36 & -90 \\ 38 & -61 & -58 & -91 \end{bmatrix}$$

```
codegen[C](submatrix,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in   r[2]
C/pretreatment:   WARNING: Unable to determine array bounds in   r[1]
C/pretreatment:   WARNING: Unable to determine array bounds in   c[2]
C/pretreatment:   WARNING: Unable to determine array bounds in   c[1]
C/pretreatment:   WARNING: Unable to determine array bounds in   r[1]
C/pretreatment:   WARNING: Unable to determine array bounds in   c[1]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[r[1]-1+i,c[1]-1+j]
#include <math.h>
#include <stdlib.h>
double submatrix(double A,double r,double c)
{
  int i;
  int j;
  int n;
  int m;
  double *B;
  {
    m = r[1]+1-r[0];
    n = c[1]+1-c[0];
    B = (double *) malloc(sizeof(double)*(m*n));
    for(i = 1;i <= m;i++)
      for(j = 1;j <= n;j++)
        B[(i-1)*n+j-1] = A[r[0]-2+i][c[0]-2+j];
C/expression:   Unknown function:   evalm   will be left as is.
    return(evalm(B));
  }
}
```

```
#include <stdlib.h>
#include<iostream.h>
#include "vecdefs.h"
#include "mvm.h"
//
MATRIX_double  submatrix(const MATRIX_double&
A,const VECTOR_double & r,
  const VECTOR_double &c)
{
  {
 int   m = r[1]+1-r[0];
 int   n = c[1]+1-c[0];
   MATRIX_double B(m,n);
   for(int i = 1;i <= m;i++)
     for(int j = 1;j <= n;j++)
       B(i-1,j-1) = A(r[0]-2+i,c[0]-2+j);
   return(B);
```

```
  }
}
```

11.4 linalg[vandermonde] - create a Vandermonde matrix

Calling Sequence:

vandermonde(L)

Parameters:

L - a list of expressions

Description:

The function vandermonde(L) returns the Vandermonde matrix formed from the elements of the list. This square matrix has as its (i,j)-th entry L[i]ˆ(j-1).

The command with(linalg,vandermonde) allows the use of the abbreviated form of this command.

Here is a version for maplelib

```
vandermonde := proc(v)
local van,n,i,j;
n := nops(v);
van := matrix(n,n);
for i from 1 to n do van[i,1]:=1:
for j from 2 to n do van[i,j] := v[i]^(j-1) od od:
evalm(van) end:
```

```
vandermonde([23,4,6,5]);
```

Ok, it works in Maple. Now export it.

```
with(codegen);
C(vandermonde,ansi);
```

Here is a modification of the exported code that compiles and tests in gcc

```
#include<math.h>
#include "vecdefs.h"
#include MATRIX_H
#include "mvm.h"
//
MATRIX_double vandermonde(const VECTOR_double  & v)
{
  int i;
  int j;
  {
 int  n = v.size();
     MATRIX_double van(n,n);
    for(i = 1;i <= n;i++)
    {
      van(i-1,0) = 1.0;
      for(j = 2;j <= n;j++)
        van(i-1,j-1) = pow(v[i-1],1.0*j-1.0);
    }
    return(van);
```

```
  }
}
```

Need a version of band, and fibonacci.

```
fibonacci(4);
```

12 Problems with Inverses.

12.1 Bidiagonal and tridiagonal matrices of ones.

```
with(linalg):
```

```
Warning, new definition for norm
Warning, new definition for trace
```

Inverses of bidiagonal matrices of ones. The bidiagonal matrix of ones is n by n and has ones down the diagonal and the superdiagonal. All other entries are 0. Here is a word to generate it.

```
bidiag := proc(n)
local B, i, j;
B := matrix(n,n);
for i from 1 to n do for j from 1 to n do
if j = 1+i or j = i then B[i,j] :=1 else B[i,j] := 0 fi od od;
evalm(B);
end:
with(codegen);
```

```
Warning, new definition for fortran
```

[*C*, *GRAD*, *GRADIENT*, *HESSIAN*, *JACOBIAN*, *cost*, *declare*, *dontreturn*, *fortran*, *horner*, *intrep2maple*, *joinprocs*, *makeglobal*, *makeparam*, *makeproc*, *makevoid*, *maple2intrep*, *optimize*, *packargs*, *packlocals*, *packparams*, *prep2trans*, *renamevar*, *split*, *swapargs*]

```
C(bidiag);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
B[i,j]
C/pretreatment:   WARNING: Unable to determine array bounds in
B[i,j]
#include <math.h>
double bidiag(n)
double n;
{
  double B;
  int i;
  int j;
  {
C/expression:   Unknown function:   matrix   will be left as is.
    B = matrix(n,n);
    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        if( j == 1.0+i || j == i )
          B[i-1][j-1] = 1.0;
        else
          B[i-1][j-1] = 0.0;
C/expression:   Unknown function:   evalm   will be left as is.
    return(evalm(B));
  }
}
```

#include <math.h> #include ”vecdefs.h” #include <mvm.h> // MATRIX_double bidiag(int n) // { MATRIX_double B(n,n); int i; int j; { for(i = 1;i <= n;i++) for(j = 1;j <= n;j++) if(j == 1.0+i —— j == i) B(i-1,j-1) = 1.0; else B(i-1,j-1) = 0.0;

return(B); } }

```
bidiag (7);
```

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
inverse(bidiag(8));
```

$$\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

12.1.1 *Problem:* *Is every bidiagonal matrix of ones invertible?*

Prove or find a counterexample.
Tridiagonal matrix of ones are defined in a similar manner.

```
tridiag := proc(n)
local B, i, j;
B := matrix(n,n);
for i from 1 to n do for j from 1 to n do
if j = i -1 or j = 1+i or j = i then B[i,j] :=1 else B[i,j] :=
0 fi od od;
evalm(B);
end:
```

```
tridiag(3);
```

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

```
inverse(tridiag(6));
```

$$\begin{bmatrix} 0 & 1 & -1 & 0 & 1 & -1 \\ 1 & -1 & 1 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 1 & -1 & 1 \\ -1 & 1 & 0 & -1 & 1 & 0 \end{bmatrix}$$

12.1.2 ***Problem.*** *Which tridiagonal matrices of ones are invertible?*

```
C(tridiag,ansi);
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
B[i,j]
C/pretreatment:   WARNING: Unable to determine array bounds in
B[i,j]
#include <math.h>
double tridiag(double n)
{
  double B;
  int i;
  int j;
  {
C/expression:   Unknown function:   matrix   will be left as is.
    B = matrix(n,n);
    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        if( j == i-1.0 || j == 1.0+i || j == i )
          B[i-1][j-1] = 1.0;
        else
          B[i-1][j-1] = 0.0;
    return(evalm(B));
  }
}
```

```
MATRIX_double tridiag(const int &n)
{
  int i;
  int j;
  {
MATRIX_double  B(n,n);
    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        if( j == i-1 —— j == 1+i —— j == i )
          B(i-1,j-1) = 1.0;
        else
          B(i-1,j-1) = 0.0;
    return(B);
  }
}
```

12.2 Hilbert matrices.

These matrices can be used to show that one must exercise care when using finite precision arithmetic to perform matrix operations. The results may be way off.

```
h6 :=hilbert(6);
```

$$h6 := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{bmatrix}$$

```
ih6:=inverse(h6);
```

$$ih6 := \begin{bmatrix} 36 & -630 & 3360 & -7560 & 7560 & -2772 \\ -630 & 14700 & -88200 & 211680 & -220500 & 83160 \\ 3360 & -88200 & 564480 & -1411200 & 1512000 & -582120 \\ -7560 & 211680 & -1411200 & 3628800 & -3969000 & 1552320 \\ 7560 & -220500 & 1512000 & -3969000 & 4410000 & -1746360 \\ -2772 & 83160 & -582120 & 1552320 & -1746360 & 698544 \end{bmatrix}$$

```
Digits := 6;
```

$$Digits := 6$$

```
eh6 := evalf(evalm(h6));
```

$$eh6 := \begin{bmatrix} 1. & .500000 & .333333 & .250000 & .200000 & .166667 \\ .500000 & .333333 & .250000 & .200000 & .166667 & .142857 \\ .333333 & .250000 & .200000 & .166667 & .142857 & .125000 \\ .250000 & .200000 & .166667 & .142857 & .125000 & .111111 \\ .200000 & .166667 & .142857 & .125000 & .111111 & .100000 \\ .166667 & .142857 & .125000 & .111111 & .100000 & .0909091 \end{bmatrix}$$

```
ieh6 :=inverse(eh6);
```

$$ieh6 := \begin{bmatrix} 11.0599 & 58.5632 & -1203.88 & 4144.57 & -5226.01 & 2226.06 \\ 58.5632 & -4338.50 & 38101.0 & -112425. & 133701. & -55342.0 \\ -1203.88 & 38101.0 & -273828. & 740738. & -840337. & 337873. \\ 4144.57 & -112425. & 740738. & -.189645\,10^{7} & .207174\,10^{7} & -810478. \\ -5226.01 & 133701. & -840337. & .207174\,10^{7} & -.219506\,10^{7} & 837382. \\ 2226.06 & -55342.0 & 337873. & -810478. & 837382. & -312216. \end{bmatrix}$$

```
map(evalm,[eh6&*ieh6,h6&*ih6]);
```

$$\left[\begin{bmatrix} .999 & -.19 & 1.3 & -1. & -1. & -1.1 \\ 0 & .91 & .9 & 1. & 0 & -.2 \\ 0 & -.05 & 1.5 & 0 & 0 & .1 \\ .001 & -.10 & .5 & 2.0 & .4 & .8 \\ -.001 & .02 & .1 & .2 & 1.2 & -.2 \\ -.001 & -.08 & -.1 & .2 & -.4 & 1.0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}\right]$$

```
Digits := 10;
```

$$Digits := 10$$

```
eh6 := evalf(evalm(h6));
```

$$\begin{aligned} & eh6 := \\ & [1., .5000000000, .3333333333, .2500000000, .2000000000, .1666666667] \\ & [.5000000000, .3333333333, .2500000000, .2000000000, .1666666667, \\ & .1428571429] \\ & [.3333333333, .2500000000, .2000000000, .1666666667, .1428571429, \\ & .1250000000] \\ & [.2500000000, .2000000000, .1666666667, .1428571429, .1250000000, \\ & .1111111111] \\ & [.2000000000, .1666666667, .1428571429, .1250000000, .1111111111, \\ & .1000000000] \\ & [.1666666667, .1428571429, .1250000000, .1111111111, .1000000000, \\ & .09090909091] \end{aligned}$$

```
ieh6 := inverse(eh6);
```

$$\begin{aligned} & ieh6 := \\ & [35.99835642, -629.9538302, 3359.692089, -7559.208458, 7559.134650, \\ & -2771.661717] \\ & [-629.9538302, 14698.70830, -88191.40873, 211657.9574, -220475.9375, \\ & 83150.60435] \\ & [3359.692089, -88191.40873, 564422.9609, -.1411053845\,10^7, \\ & .1511840609\,10^7, -582057.8112] \\ & [-7559.208458, 211657.9574, -.1411053845\,10^7, .3628425849\,10^7, \\ & -.3968592256\,10^7, .1552161002\,10^7] \\ & [7559.134650, -220475.9375, .1511840609\,10^7, -.3968592256\,10^7, \\ & .4409555888\,10^7, -.1746186894\,10^7] \\ & [-2771.661717, 83150.60434, -582057.8111, .1552161002\,10^7, \\ & -.1746186894\,10^7, 698476.5498] \end{aligned}$$

```
evalm(eh6&*ieh6);
```

$$\begin{bmatrix} .9999994 & -.00001 & -.00010 & .0001 & -.0001 & .0001 \\ -.5\,10^{-6} & 1.00000 & 0 & 0 & .0002 & -.00001 \\ -.3\,10^{-6} & -.00002 & 1.00001 & .0001 & 0 & .00023 \\ 0 & -.3\,10^{-5} & -.00002 & 1.0000 & .0001 & .00003 \\ 0 & -.6\,10^{-5} & .00009 & -.0001 & 1.0000 & .00008 \\ 0 & -.1\,10^{-5} & .00005 & 0 & 0 & 1.00006 \end{bmatrix}$$

This is much better, but now check out hilbert(10).

```
evalm(evalf(hilbert(10))&*inverse(evalf(hilbert(10))));
```

$$\begin{bmatrix} 1.0024 & .10 & -1.1 & 4. & 0 & 30. & 180. & -80. & -50. & -29. \\ .0005 & 1.04 & -.7 & -6. & 90. & 40. & 140. & -10. & -150. & 0 \\ .0005 & .05 & .1 & 3. & -10. & 20. & 90. & 30. & -40. & -4. \\ .0004 & .09 & -.8 & 2. & 50. & -10. & 70. & -110. & 50. & 2. \\ 0 & .02 & -.4 & 3. & 10. & 80. & 160. & 20. & -50. & -8. \\ .0002 & 0 & .3 & 13. & 40. & 20. & 180. & -50. & -30. & 11. \\ .0007 & .03 & .8 & -7. & 30. & 70. & -10. & 50. & -10. & 14. \\ .0002 & .03 & -.3 & 11. & -40. & 80. & 70. & -50. & 20. & 13. \\ 0 & .01 & -1.0 & -5. & -9. & 50. & 90. & 30. & -90. & -6. \\ .0004 & .02 & .2 & -5. & -8. & -40. & 60. & -30. & -140. & 19. \end{bmatrix}$$

```
hilbert := proc(n)
local i,j,A;
A := array(1..n,1..n);
for i from 1 to n do
for j from 1 to n do
A[i,j] := 1/(i+j-1); od od;
evalm(A); end;
```

```
hilbert := proc(n)
    local i, j, A;
        A := array(1..n, 1..n);
        for i to n  do for j to n do A_{i, j} := 1/(i + j - 1) od od;
        evalm(A)
    end
```

```
hilbert(5);
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

```
   with(codegen):

Warning, new definition for fortran
   C(hilbert,ansi);

#include <math.h>
#include <stdlib.h>
double hilbert(int n)
{
  int i;
  int j;
  double *A;
  {
    A = (double *) malloc(sizeof(double)*(n*n));
    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        A[(i-1)*n+j-1] = 1/(i+j-1.0);
C/expression:   Unknown function:   evalm   will be left as is.
    return(evalm(A));
  }
}
```

```
MATRIX_double hilbert(int n)
{
  int i;
  int j;
 //
  {
  MATRIX_double  A(n,n) ;
    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        A(i-1,j-1) = 1/(i+j-1.0);
 //
    return(A);
  }
}
```

13 Gaussian Elimination with good record keeping.

13.1 The Naive case: No row swapping allowed

A elimination matrix is a square matrix with 1's down the diagonal and 0's elsewhere except at one entry below the diagonal. Here is a word to generate an elimination matrix.

```
restart;with(linalg):
```

```
Warning, new definition for norm
Warning, new definition for trace
```

```
elim := proc(size,row,col,entry)
local A,i;
A := diag(seq(1,i = 1 ..  size));
A[row,col] := entry;
evalm(A)
end:
elim(3,3,2,9);
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 9 & 1 \end{bmatrix}$$

Let's export to C++

```
with(codegen):
```

```
Warning, new definition for fortran
```

```
C(elim,ansi):
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
A[row,col]
#include <math.h>
double elim(double size,int row,int col,double entry)
{
  double A;
  double i;
  {
C/expression:   Unknown function:   diag   will be left as is.
C/expression:   Unknown function:   seq   will be left as is.
C/expression:   Unable to translate   i = 1 .. size   into C
C/expression:   Replacing value with three parameters   i, 1, size
    A = diag(seq(1.0,i,1.0,size));
    A[row-1][col-1] = entry;
C/expression:   Unknown function:   evalm   will be left as is.
    return(evalm(A));
  }
}
```

```
// The elimination matrix of size size x size
// with entry entry in row row and column col.
#include <math.h>
#include ”vecdefs.h”
#include MATRIX_H
//
MATRIX_double elim(int size,int row,int col,double
```

```
entry)
{
MATRIX_double A(size,size);
  int i;
  int j;
  {
     for(i = 1;i <= size;i++)
      for(j = 1;j <= size;j++)
        if( i == j )
          A(i-1,j-1) = 0.1E1;
        else
          A(i-1,j-1) = 0.0;
    A(row-1,col-1) = entry;
    return(A);
  }
}
```

1 ***The LU factorization theorem:*** *If A is a square matrix which can be reduced to upper triangular form without row interchanges so that no pivot is zero, then A can be factored as the product of a lower unit triangular matrix with an upper triangular matrix with nonzero diagonal entries.*

Look at an example. Generate a random 3 by 3 matrix.

```
A := randmatrix(3,3);
```

$$A := \begin{bmatrix} -91 & -47 & -61 \\ 41 & -58 & -90 \\ 53 & -1 & 94 \end{bmatrix}$$

Now the first elimination matrix looks like this –

```
e21 := elim(3,2,1,-A[2,1]/A[1,1]);
```

$$e21 := \begin{bmatrix} 1 & 0 & 0 \\ \frac{41}{91} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiply E21 by A to make the 21 entry 0.

```
U := evalm(e21 &* A);
```

$$U := \begin{bmatrix} -91 & -47 & -61 \\ 0 & \frac{-7205}{91} & \frac{-10691}{91} \\ 53 & -1 & 94 \end{bmatrix}$$

```
e31 := elim(3,3,1,-U[3,1]/U[1,1]);
```

$$e31 := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{53}{91} & 0 & 1 \end{bmatrix}$$

```
U := evalm(e31 &* U);
```

$$U := \begin{bmatrix} -91 & -47 & -61 \\ 0 & \frac{-7205}{91} & \frac{-10691}{91} \\ 0 & \frac{-2582}{91} & \frac{5321}{91} \end{bmatrix}$$

```
e32 := elim(3,3,2,-U[3,2]/U[2,2]);
```

$$e32 := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \frac{-2582}{7205} & 1 \end{bmatrix}$$

```
U := evalm(e32 &* U);
```

$$U := \begin{bmatrix} -91 & -47 & -61 \\ 0 & \frac{-7205}{91} & \frac{-10691}{91} \\ 0 & 0 & \frac{724637}{7205} \end{bmatrix}$$

So at this point, we have that e32 e31 e21 A = U. To undo the eliminations, we only need to multiply U on the left successively by the inverse of e32, then e31, then e21 to get A back again. Note that the inverse of elimin(size,row,col,entry) is just the elimination matrix elim(size,row,col,-entry).

```
evalm(elim(5,3,2,entry)&*elim(5,3,2,-entry));
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
L :=
evalm(inverse(e21)&*inverse(e31)&*inverse(e32));
```

$$L := \begin{bmatrix} 1 & 0 & 0 \\ \frac{-41}{91} & 1 & 0 \\ \frac{-53}{91} & \frac{2582}{7205} & 1 \end{bmatrix}$$

Note that the product of the elimination matrices in reverse order is just the lower triangular matrix whose entries below the diagonal are the negatives of the multiplier entries of the elimination matrices below the diagonal.

Here is a word to factor a matrix into the product of a lower triangular and upper triangular matrix.

```
lufact := proc(A,‘L‘,‘U‘)
local k,i,j,s,p,el;
k := rowdim(A);
U := evalm(A);
L := diag(seq(1,i = 1 ..  k));
for i to k-1 do
```

```
if U[i,i] = 0 then
ERROR(`bad step `,i)
else
for j from i+1 to k do
el := elim( k,j,i,-U[j,i]/U[i,i] );
L[j,i] := U[j,i]/U[i,i];
U := evalm(el &* U)
od
fi
od;
if U[k,k] = 0 then
ERROR(`0 pivot`,i) fi;
NULL:
end:
```

We can check this out on some random 4 by 4 matrices.

```
A := randmatrix(4,4):  lufact(A ,L,U):
map(evalm,[A,L,U,L&*U]);
```

$$\left[\begin{bmatrix}83&-86&23&-84\\19&-50&88&-53\\85&49&78&17\\72&-99&-85&-86\end{bmatrix},\begin{bmatrix}1&0&0&0\\\frac{19}{83}&1&0&0\\\frac{85}{83}&\frac{-11377}{2516}&1&0\\\frac{72}{83}&\frac{2025}{2516}&\frac{-431597}{1078261}&1\end{bmatrix},\begin{bmatrix}83&-86&23&-84\\0&\frac{-2516}{83}&\frac{6867}{83}&\frac{-2803}{83}\\0&0&\frac{1078261}{2516}&\frac{-125005}{2516}\\0&0&0&\frac{-6295965}{1078261}\end{bmatrix},\right.$$
$$\left.\begin{bmatrix}83&-86&23&-84\\19&-50&88&-53\\85&49&78&17\\72&-99&-85&-86\end{bmatrix}\right]$$

After doing a few of these, you begin to wonder what the probability is that a random 4 by 4 matrix cannot be reduced to upper triangular form with a full set of (nonzero) pivots.

Sometimes they do, and in this case we need to interchange a row lower down with what would have been the pivot row if the pivot entry were nonzero. Permutation matrices perform this operation. In the next section, we build this into the algorithm.

But first, let's export lufact to C++.

```
C(lufact,ansi):
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
U[i,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[i,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
L[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[i,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
```

```
U[k,k]
#include <math.h>
void lufact(double A,double *L,double *U)
{
  int k;
  int i;
  int j;
  double s;
  double p;
  double el;
  {
    k = rowdim(A);
    *U = evalm(A);
C/expression:   Unable to translate   i = 1 .. k   into C
C/expression:   Replacing value with three parameters   i, 1, k
    *L = diag(seq(1.0,i,1.0,k));
    for(i = 1;i <= k-1.0;i++)
      if( U[i-1][i-1] == 0.0 )
        {
          fprintf(stderr, "bad step i" );
          exit(1);
        }
      else
        for(j = i+1;j <= k;j++)
        {
          el = elim(k,j,i,-U[j-1][i-1]/U[i-1][i-1]);
          L[j-1][i-1] = U[j-1][i-1]/U[i-1][i-1];
          *U = evalm(&*(el,U));
        }
    if( U[k-1][k-1] == 0.0 )
      {
        fprintf(stderr, "0 pivoti" );
        exit(1);
      }
    return;
  }
}
```

After massaging a little, exported code looks like this.

```
#include <math.h>
#include <stdlib.h>
 #include<iostream.h>
#include ”vecdefs.h”
#include ”mvm.h”
//
int rowdim(const MATRIX_double & );
MATRIX_double elim(int,int,int,double);
 MATRIX_double operator*(const MATRIX_double &,const MATRIX_double
&);
void lufact(const MATRIX_double &A ,MATRIX_double
& L,MATRIX_double & U)
{
  int k;
  int i;
  int j;
```

```
  {
   k = rowdim(A);
   MATRIX_double el(k,k);
   for(i = 1;i<=k-1;i++)  U(i,i)=1;
   for(i = 1;i <= k-1;i++)
     if( U(i-1,i-1) == 0.0 )
       {
  //        fprintf(stderr, ”bad stepi” );
         exit(1);
       }
     else
       for(j = i+1;j <= k;j++)
       {
         el = elim(k,j,i,-U(j-1,i-1)/U(i-1,i-1));
         L(j-1,i-1) = U(j-1,i-1)/U(i-1,i-1);
         U = el*U;
       }
   if( U(k-1,k-1) == 0.0 )
     {
    //   fprintf(stderr, ”0 pivoti” );
       exit(1);
     }
  }
}
```

13.2 Row interchanges: The PA=LU Theorem

Permutation Matrices are square matrices with exactly one 1 in each row and column and 0's elsewhere. They obtained from the identity matrix by permuting its rows. The simplest type is the transposition matrix, which is obtained by interchanging two rows of the identity matrix. Here are some words that are useful in generating permutation matrices. The first word

```
swaprow := proc(A,row1,row2)
local i,n,s;
n := ‘linalg/coldim‘(A);
for i from 1 to n do
s := A[row1,i]:
A[row1,i]:= A[row2,i]:
A[row2,i]:=s:
od:
A end:
A := linalg[randmatrix](4,4);
```

$$A := \begin{bmatrix} -32 & 78 & 39 & 94 \\ 68 & -17 & -98 & -36 \\ 40 & 22 & 5 & -88 \\ -43 & -73 & 25 & 4 \end{bmatrix}$$

```
evalm(swaprow(A,2,3));
```

$$\begin{bmatrix} -32 & 78 & 39 & 94 \\ 40 & 22 & 5 & -88 \\ 68 & -17 & -98 & -36 \\ -43 & -73 & 25 & 4 \end{bmatrix}$$

This word creates the n x n identity matrix.

```
ident := proc(n)
local i,j,A;
A := MATRIX(n,n):
for i from 1 to n do for j from 1 to n do
A[i,j]:=0:  if i=j then A[i,i]:=1 fi od
od;
A end:
evalm(ident(4));
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This word creates a permutation matrix from the size x size identity matrix by interchanging row1 with row2.

```
interchange := proc(size,row1,row2)
local A,i;
A := ident(size);
A := swaprow(A,row1,row2);
evalm(A)
end:
evalm(interchange(3,3,2));
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

```
C(swaprow,ansi):
C(ident,ansi):
C(interchange,ansi):
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
A[row1,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[row1,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[row2,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[row2,i]
#include <math.h>
double swaprow(double A,int row1,int row2)
{
  int i;
```

```
  double n;
  double s;
  {
    n = linalg/coldim(A);
    for(i = 1;i <= n;i++)
    {
      s = A[row1-1][i-1];
      A[row1-1][i-1] = A[row2-1][i-1];
      A[row2-1][i-1] = s;
    }
    return(A);
  }
}
C/pretreatment:   WARNING: Unable to determine array bounds in
A[i,j]
C/pretreatment:   WARNING: Unable to determine array bounds in
A[i,i]
#include <math.h>
double ident(double n)
{
  int i;
  int j;
  double A;
  {
    A = MATRIX(n,n);
    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
      {
        A[i-1][j-1] = 0.0;
        if( i == j )
          A[i-1][i-1] = 1.0;
      }
    return(A);
  }
}
#include <math.h>
double interchange(double size,double row1,double row2)
{
  double A;
  double i;
  {
    A = ident(size);
    A = swaprow(A,row1,row2);
    return(evalm(A));
  }
}
```

Here is the modified code.

```
#include<iostream.h>
#include ”vecdefs.h”
#include ”mvm.h”
//
int rowdim(const MATRIX_double & );
MATRIX_double elim(int,int,int,double);
 MATRIX_double operator*(const MATRIX_double &,const MATRIX_double
&);
//
```

```
void swaprow(MATRIX_double &A,int row1,int row2)
{
 int i;
 double n;
 double s;
int coldim(const MATRIX_double & );
 {
  n = coldim(A);
  for(i = 1;i <= n;i++)
  {
   s = A(row1-1,i-1);
   A(row1-1,i-1) = A(row2-1,i-1);
   A(row2-1,i-1) = s;
  }
 }
}
//
MATRIX_double ident(int n)
{
 int i;
 int j;
  {
  MATRIX_double A(n,n);
  for(i = 1;i <= n;i++)
     for(j=1;j<=n;j++) {A(i-1,j-1)=0; if (i==j)
A(i-1,j-1)=1;}
  return(A);
 }
}
//
MATRIX_double interchange(int size,int row1,int row2)
{
MATRIX_double A(size,size);
 double i;
 {
  A = ident(size);
  swaprow(A,row1,row2);
  return(A);
 }
}
```

If a zero pivot occurs, then we want to search down the pivot column until a nonzero entry (to our liking) is found and do a row interchange p on U to bring that entry up to the pivot row. We can, making use of a clever trick, arrange things so that the production of all the row interchanges needed can be done at the beginning. That's the statement of

2 ***The*** *PA=LU factorization theorem:* *If a square matrix A is invertible, then a permutation matrix P can be found so that PA can be factored into l u.*

Indication of proof: In the reduction of A to upper triangular form, we multiply on the left by elimination matrices and interchange matrices. So suppose we are at the ith stage and a row interchange p_{ij} is needed. A has been reduced to $U = E\,P_i\,A$ where E is a product of elimination matrices with nonzero entries below the diagonal and to the left of the ith column, and P_i is product of the row interchanges needed (if any) needed up to the ith step. The trick is to notice that $p_{ij}\,E = p_{ij}\,E\,p_{ij}\,p_{ij}$ because $p_{ij}\,p_{ij}$ is the identity matrix, and $p_{ij}\,E\,p_{ij}$is a lower triangular matrix that looks just like E except that the multipliers in the ith and jth rows have been interchanged. So write

$p_{ij}\,U = p_{ij}\,E\,P_i\,A$ which can be rewritten $\{p_{ij}\,E\,p_{ij}\}\,\{p_{ij}\,P_i\,A\} = E\,P_{i+1}\,A$. Continue this for n-1 steps where A is an n by n matrix. At this point we have $U = E\,P\,A$, so $P\,A = L\,U$, where L is the inverse of E.

13.2.1 ***Problem.*** *Modify the definition of lufact to get a plufact(‘P‘,A,‘L‘,‘U‘).*

There are words in linalg to do gaussian elimination with row interchanges on an augmented matrix.

```
A := randmatrix(3,4);
```

$$A := \begin{bmatrix} 1 & 30 & 81 & -5 \\ -28 & 4 & -11 & 10 \\ 57 & -82 & -48 & -11 \end{bmatrix}$$

```
gausselim(A);
```

$$\begin{bmatrix} 1 & 30 & 81 & -5 \\ 0 & 844 & 2257 & -130 \\ 0 & 0 & \frac{26821}{211} & \frac{-426}{211} \end{bmatrix}$$

```
backsub(%);
```

$$\left[\frac{-9839}{26821}, \frac{-2992}{26821}, \frac{-426}{26821}\right]$$

This gives the solution to the original system.

```
plufact := proc(‘P‘,A,‘L‘,‘U‘ )
local k,i,j,t,el,p,l ;
k := rowdim(A);
U := evalm(A);
L := diag(seq(1,i = 1 ..  k));
P := diag(seq(1,i = 1 ..  k));
for i to k-1 do
if U[i,i] = 0 then
for t from i+1 to k while U[t,i]=0 do od ;
if U[t,i] <> 0 then p := interchange(k,t,i);
P := evalm( p &*P);
U := evalm(p &*U);
L := evalm(p&*L&*p)
else
```

```
RETURN(ERROR(cat( ‘zero pivot encountered at ‘,i,‘ step ‘))fi fi;
for j from i+1 to k do
if not U[j,i] = 0 then
L[j,i] := U[j,i]/U[i,i];
el := elim( k,j,i,-L[j,i] );
U := evalm(el &* U); fi;
od;
od;
if U[k,k] = 0 then
RETURN( ERROR(cat( ‘zero pivot encountered at ‘,i,‘ step ‘)) fi;
NULL:
end:
plufact(P,A,L,U);
evalf(map(evalm,[P,A,L,U]));
```

$$\left[\begin{bmatrix} 1. & 0 & 0 \\ 0 & 1. & 0 \\ 0 & 0 & 1. \end{bmatrix}, \begin{bmatrix} 1. & 30. & 81. & -5. \\ -28. & 4. & -11. & 10. \\ 57. & -82. & -48. & -11. \end{bmatrix}, \begin{bmatrix} 1. & 0 & 0 \\ -28. & 1. & 0 \\ 57. & -2.123222749 & 1. \end{bmatrix},\right.$$
$$\left.\begin{bmatrix} 1. & 30. & 81. & -5. \\ 0 & 844. & 2257. & -130. \\ 0 & 0 & 127.1137441 & -2.018957346 \end{bmatrix}\right]$$

Here's the exported code

```
with(codegen):
```

```
Warning, new definition for C
```

```
C(plufact,ansi):
```

```
C/pretreatment:   WARNING: Unable to determine array bounds in
U[i,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[t,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[t,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
L[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[i,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
L[j,i]
C/pretreatment:   WARNING: Unable to determine array bounds in
U[k,k]
#include <math.h>
double plufact(double *P,double A,double *L,double *U)
{
  int k;
  int i;
  int j;
  int t;
```

```
  double el;
  double p;
  double l;
  {
    k = rowdim(A);
    *U = evalm(A);
C/expression:   Unable to translate   i = 1 .. k   into C
C/expression:   Replacing value with three parameters   i, 1, k
    *L = diag(seq(1.0,i,1.0,k));
C/expression:   Unable to translate   i = 1 .. k   into C
C/expression:   Replacing value with three parameters   i, 1, k
    *P = diag(seq(1.0,i,1.0,k));
    for(i = 1;i <= k-1.0;i++)
    {
      if( U[i-1][i-1] == 0.0 )
        {
          for(t = i+1;0;t++);
          if( U[t-1][i-1] != 0.0 )
            {
              p = interchange(k,t,i);
              *P = evalm(&*(p,P));
              *U = evalm(&*(p,U));
              *L = evalm(&*(&*(p,L),p));
            }
          else
            return(Error(cat(zero pivot encountered at ,i, step )));
        }
      for(j = i+1;j <= k;j++)
        if( !(U[j-1][i-1] == 0.0) )
          {
            L[j-1][i-1] = U[j-1][i-1]/U[i-1][i-1];
            el = elim(k,j,i,-L[j-1][i-1]);
            *U = evalm(&*(el,U));
          }
    }
    if( U[k-1][k-1] == 0.0 )
      return(Error(cat(zero pivot encountered at ,i, step )));
    return;
  }
}
```

Here is the modified exported code

```
void plufact(MATRIX_double &P,const MATRIX_double
&A,MATRIX_double &L,MATRIX_double &U)
{
  int k;
  int i;
  int j;
  int t;
  double l;
  {
    k = rowdim(A);
    U = A;
    P = ident(k);
    L = ident(k);
    MATRIX_double p(k,k);
```

```
    MATRIX_double el(k,k);
    for(i = 1;i <= k-1;i++)
    {
      if( U(i-1,i-1) == 0.0 )
        {
          for(t = i+1;0;t++);
          if( U(t-1,i-1) != 0.0 )
            {
              p = interchange(k,t,i);
               P = p*P;
               U = p*U;
               L = p*L*p;
            }
          else
            exit(1);
        }
      for(j = i+1;j <= k;j++)
        if( !(U(j-1,i-1) == 0.0) )
          {
            L(j-1,i-1) = U(j-1,i-1)/U(i-1,i-1);
            el = elim(k,j,i,-L(j-1,i-1));
             U = el*U;
          }
    }
    if( U(k-1,k-1) == 0.0 )
      exit(1);
  }
}
```

Let's test this on some matrices where row interchanges have to take place.

```
n := 3 :  A := randmatrix(n,n);
```

$$A := \begin{bmatrix} -51 & -73 & -73 \\ -91 & 1 & 5 \\ -86 & 43 & -4 \end{bmatrix}$$

```
n,op( map(op,convert(A,listlist)));
```

$$3, -51, -73, -73, -91, 1, 5, -86, 43, -4$$

```
plufact(p,A,l,u);
evalf(map(evalm,[A,l,u,p]));
```

$$\left[\begin{bmatrix} -51. & -73. & -73. \\ -91. & 1. & 5. \\ -86. & 43. & -4. \end{bmatrix}, \begin{bmatrix} 1. & 0 & 0 \\ 1.784313725 & 1. & 0 \\ 1.686274510 & 1.265461607 & 1. \end{bmatrix}, \begin{bmatrix} -51. & -73. & -73. \\ 0 & 131.2549020 & 135.2549020 \\ 0 & 0 & -52.06184643 \end{bmatrix}, \right.$$

$$\begin{bmatrix} 1. & 0 & 0 \\ 0 & 1. & 0 \\ 0 & 0 & 1. \end{bmatrix}\Bigg]$$

```
evalm(p&*A-l&*u);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
n,op( map(op,convert(A,listlist)));
```

$$3, -51, -73, -73, -91, 1, 5, -86, 43, -4$$

Here is the output from feeding this to plufact.

```
-51 -73 -73
-91 1 5
-86 43 -4
1 0 0
0 1 0
0 0 1
1 0 0
1.78431 1 0
1.68627 1.26546 1
-51 -73 -73
0 131.255 135.255
0 0 -52.0618
```

Index