AUTODIF

A C++ Array Language Extension with Automatic Differentiation For Use in Nonlinear Modeling and Statistics

admb-project.org

Licence

Contents

Chapter 1

Introduction

What is AUTODIF?

AUTODIF is an array language extension to C++ which allows the user to automatically calculate the partial derivatives of a function of many independent variables by simply redeclaring the usual C floating point types doubles and arrays of doubles to be the corresponding AUTODIF type, `dvariable` and `dvar_vector`. The user can write the code for the function using any valid C++ syntax. AUTODIF does not impose any restrictions on allowable code constructions. Using AUTODIF the programmer can manipulate vectors and matrices as single objects in complicated mathematical calculations, at the same time obtaining derivatives of the resulting function with respect to the independent variables.

When calculating derivatives there is no restriction on calling user supplied functions which may themselves call other functions, so long as the types have been redefined to be the corresponding AUTODIF types. Since C++ is upwardly compatible with ANSI C, existing numerical routines written in C can be easily modified to use AUTODIF.

Who should use AUTODIF? Anyone who would like to have the derivatives of a complicated function computed automatically without any additional programming effort will find AUTODIF useful. It is particularly useful for optimization problems involving a differentiable function of many independent variables. Such problems occur for example in nonlinear statistical modeling, i.e. in nonlinear parameter estimation, in sensitivity analysis, and in finding roots of systems of equations.

Why is automatic differentiation better than estimating the derivatives by finite differences?

There are two reasons why the AUTODIF method is better than finite differences for calculating partial derivatives.

1. With AUTODIF the derivatives are calculated to the limit of machine accuracy, that is to the same degree of accuracy as the function itself. The derivatives are obtained as accurately as they would be if an analytical expression for the derivatives were evaluated on the computer. With finite differences the accuracy of the derivative calculations seldom approaches that of the limit of the machine accuracy for calculating the derivatives. Consequently, optimization schemes based on automatic differentiation tend to be more stable and perform better than those which use finite difference approximations.

2. To obtain estimates of the partial derivatives by finite difference approximations the amount of calculation necessary is proportional to the number of independent variables times the amount of time needed to calculate the function itself. With AUTODIF the amount of calculation is less than 5 times the amount of calculation needed to calculate the function itself (Griewank and Corliss 1991). In particular, the computing time does not depend on the number of independent variables. This makes AUTODIF particularly attractive for calculating derivatives for functions with many independent variables.

Is it necessary to learn C++before one can use AUTODIF?

Any experienced C programmer can learn to use AUTODIF in a few hours. While it is necessary to have a C++ compiler to use AUTODIF, only a small number of extra concepts besides those used in the C language need to be learned. This is due to the fact that C++ is essentially a superset of C, so that valid ANSI-compliant C code needs only a very small amount of modification to be converted to valid C++ code.

We feel that the best way to get a feel for how AUTODIF is used is to study a series of examples. These examples have been chosen to illustrate the use of AUTODIF on problems in different fields from engineering to statistical analysis.

How does automatic differentiation work?

Every computer program for calculating a function value, no matter how complicated, can be broken down into a series of binary and unary mathematical operations. Automatic differentiation is simply the chain rule from elementary calculus applied to this series. The following simple example illustrates the chain rule.

$$\begin{aligned} u &= x^2 + y^2 \\ v &= x + 3y \\ f &= \sin(u) + \exp(v) \qquad (1.1) \end{aligned}$$

We want to calculate the derivatives of the function f with respect to the variables x and y. By the chain rule

$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial u}\frac{\partial u}{\partial x}+\frac{\partial f}{\partial v}\frac{\partial v}{\partial x} \qquad \frac{\partial f}{\partial y}=\frac{\partial f}{\partial u}\frac{\partial u}{\partial y}+\frac{\partial f}{\partial v}\frac{\partial v}{\partial y}$$

Now

$$\begin{array}{cc} \frac{\partial f}{\partial u}=\cos(u) & \tfrac{\partial f}{\partial v}= \quad \exp(v) \\ \frac{\partial u}{\partial x}=2x \qquad \frac{\partial v}{\partial x}=1 & \frac{\partial u}{\partial y}=2y \qquad \frac{\partial v}{\partial y}=3 \end{array} \tag{1.2}$$

so that

$$\frac{\partial f}{\partial x}=\cos(u)*2x+\ \exp(v)*1 \qquad \frac{\partial f}{\partial y}=\cos(u)*2y+\ \exp(v)*3$$

That is really all there is to it! Your program is broken down into a series of simple operations like these so that the derivatives can be calculated. The important point is that no matter how complicated the mathematical expressions are they can be thought of as consisting of a long series of binary and unary operations.

What is adjoint (precompiled) derivative code and why is it so important?

While the reverse mode of automatic differentiation is extremely efficient in terms of the number of calculations required to compute derivatives, the method requires a large amount of temporary storage. Each arithmetic operation used to calculate the original function generates 28–30 bytes of temporary storage. Consider that to invert a 100 by 100 matrix requires 10^6 operations. This would generate 28–30 megabytes of temporary storage requirements. Clearly it would be impractical to carry out large calculations under these conditions.

The use of precompiled derivative code combines the best of two worlds, so to speak. The small number of arithmetic operations required by the reverse mode for calculating derivatives is retained while at the same time the amount of temporary storage required is greatly reduced. The amount of storage required for inverting a 100 by 100 matrix inverse is reduced to about 500K. Even better, the amount of temporary storage requirement generated by adding two matrices is about 32 bytes no matter how large the matrices are.

Precompiled derivative code means that the actual derivative calculations are written in C++ functions. Following object oriented design considerations the precompiled code is completely "encapsulated" so that it is invisible to the user. This means that it makes no difference to the user whether a particular function has a precompiled derivative code component written for it. At the user level everything looks the same, except that performance is greatly enhanced. The use of precompiled derivative code produces the ultimate in reuseable code. For example, writing good code for the derivative of the the inverse of a matrix is not a trivial procedure, but once it has been written any user of AUTODIF can take advantage of it simply by taking the inverse of a matrix object with code like

```
M1=inv(M);  // M1 and M are dvar_matrix objects
```

AUTODIF combines array and matrix classes with the use of precompiled derivative code for common array and matrix calculations and assignments to produce a superior environment for efficient derivative calculations.

While AUTODIF has been supplied with all the computational power described above, it has been designed so that at the user level it appears like enhanced C code or perhaps as enhanced C code with an adittional array language. Before illustrating some of AUTODIF's array language facilities we will begin with a simple C code example.

The following C code for the function `fcomp` calculates the sum of squares

$$\sum_{i=0}^{n-1}(x_i-1)^2$$

This code has been written in C without using any C++ or AUTODIF extensions to show how C code can be modified to use AUTODIF for calculating derivatives.

```
#include <math.h>
double fcomp(int n,double * x)
{
  double z;
  double tmp;
  int i;
  z=0;
  for (i=0;i<n;i++)
  {
    tmp=x[i]-1;
    z=z+tmp*tmp;
  }
  return(z);
}
```

`fcomp` computes the sum of squares of n double precision floating point numbers which are contained in an array of length `n` pointed to by `x`. The expression `x[i]` refers to the i'th element of the array. (Note that the default construction for C begins indexing an array with 0.)

The arithmetic in the function `fcomp` is carried out using objects of type `double`, that is, eight byte floating point numbers. In order to calculate derivatives at the same time as the function is being calculated, the class `double` is replaced with a new AUTODIF class, the `dvariable`. An object of type `dvariable` "appears" to the user to be an object of type `double` in that it can be used in arithmetic and mathematical expressions, and it has a numerical value which can be displayed. The difference between an object of type `double` and an object of type `dvariable` is that when arithmetic is done using a `dvariable` object, information necessary for the calculation of derivatives is automatically generated as well.

In addition to objects of type `dvariable`, we shall need a class of "vector" object corresponding to `x` which generates derivative information. In AUTODIF this class is the `dvar_vector` class.

The following listing is the `fcomp` function modified to use automatic differentiation.

```
// file: fcomp_s.cpp
#include <fvar.hpp>
double fcomp(int n, dvar_vector x)
{
  dvariable z;
  dvariable tmp;
  int i;
  z=0;
  for (i=1;i<=n;i++)
  {
    tmp=x[i]-1;
    z=z+tmp*tmp;
  }
  return(value(z));
}
```

There are five changes.

1. The line `#include <fvar.hpp>` has been included. The file `fvar.hpp` contains information about AUTODIF for the C++compiler.

2. The array of variables `x` has been redefined to be of type `dvar_vector`. This AUTODIF type is like an array of doubles for which derivative information is collected. Besides providing for the calculation of derivatives, the use of the `dvar_vector` class for `x` provides an additional improvement. With the original C code there was no way of determining in the subroutine `fcomp` whether enough memory had actually been allocated to the array `x`. AUTODIF provides optional index bounds checking for all container classes such as `dvar_arrays`. This greatly speeds up program development.

3. The variables `z` and `tmp` have been declared to be of type `dvariable`. The `dvariable` type is the fundamental type for automatic differentiation. It corresponds to the C floating point type `double`.

4. The function value has been included in the line return(value(z)); The function value() returns the numerical part of an dvariable. This is necessary to convert the dvariable to a double so that the value can be returned and used in subsequent calculations (such as in a function minimizer) without generating any more derivative calculations.

5. The range of the index of the for loop has been changed to begin at 1 and include n. While this change is not strictly necessary fo this example, AUTODIF allows the user to declare arrays which have arbitrary valid index ranges. This facility often contributes to the production of simnpler looking code.

The form of the interface of the function fcomp with the rest of the program will depend on the user's application. This section illustrates two applications. The first example shows how to simply calculate the derivatives so that they can be used for any other purpose whatsoever. The second example shows how to invoke a function minimizer, supplied with AUTODIF to find the minimum of the function fcomp. In any event it is necessary to identify the independent variables for AUTODIF (these are the variables for which the partial derivatives are calculated). The independent variables are identified by declaring them to be of type independent_variables. The declaration independent_variables x(1,nvar) creates an array of 20 independent variables and identifies the independent variables for AUTODIF. An object of type gradient_structure must be declared and remain in scope until all the derivative calculations have been finished.

```
#include <math.h>
#include <fvar.hpp>
double fcomp(dvar_vector); // Function prototype declaration
void main()
{
  double f;
  int i;
  int nvar=20;
  independent_variables x(1,nvar);  // Identify the independent variables
  dvector g(1,nvar);  // Holds the vector of partial derivatives (the gradient)
  gradient_structure gs;    // must declare this structure to manage derivative
                            // calculations
  f=fcomp(nvar,x);
  gradcalc(nvar,g);        // The derivatives are calculated
  cout <<" The gradient vector is\n"<<g<<"\n"; // Print out the derivatives
}                                              // on the screen
```

Notice that although x has been declared to be an object of type independent_variables in the main routine, it is declared to be an object of type dvar_vector in fcomp. While this might appear to be an error it is actually an important part of the interface. It is necessary to identify both what the independent variables are, and when one should begin to collect derivative information for calculations involving them. When the vector x is passed to the function fcomp, the compiler will invoke a constructor to convert the type independent_variables to the type dvar_vector. At this time the steps necessary for identifying the independent variables and initializing the collection of information needed for calculating the derivatives are carried out. With the Borland C++ compiler a warning message

```
Temporary used for type xxx in function yyy
```

is printed out notifying the user that a constructor has been invoked to perform the necessary conversion. Such a mesage is usually benign and we tend to ignore them since we have tried to design AUTODIF so that unwanted type conversions do not occur – however it may indicate that some unintended conversion has taken place. This is one of the subtler pitfalls of C++ programming. If you want to avoid such warning messages it is possible to explicitly "cast" the object `x` to the desired type as

```
  f=fcomp(nvar,dvar_vector(x));
```

The next example shows how the above code is modified to invoke the function minimizer routine. The function to be minimized and the function minimizer are embedded in a loop so that it is not necessary for the user's function to call the function minimizer or for the function minimizer to call the user's function. This eliminates the need to communicate with either routine by passing global variables. Since the code which sets up the loop is independent of the user's function to be minimized we have defined two macros, `BEGIN_MINIMIZATION` and `END_MINIMIZATION` which will expand to produce the loop structure. The first macro, `BEGIN_MINIMIZATION` takes five arguments, the number of variables in the function to be minimized, the name of the vector of independent variables, the name of the dependent variable, the name of the vector of partial derivatives and the name of the `fmm` control structure. It is the users responsibility to declare these five objects. The second macro, `END_MINIMIZATION` takes two arguments, the number of variables and the name of the vector of partial derivatives.

```
//file: simple.cpp
#include <fvar.hpp>

double fcomp(int, dvar_vector); // Function prototype declaration

#ifdef __BCPLUSPLUS__
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  long _stack = 20000;
#endif
void main()
{
  int nvar=20; // This is the number of independent variables
  independent_variables x(1,nvar); // these are the independent variables
  double f;  // This is the dependent variable
  dvector g(1,nvar);  // Holds the vector of partial derivatives (the gradient)
  fmm fmc(nvar);      // Create structure to manage minimization
  BEGIN_MINIMIZATION(nvar,f,x,g,fmc) // Macro to set up beginning of
                                     // minimization loop
    f=fcomp(nvar,x);
  END_MINIMIZATION(nvar,g)    // Macro to set up end of minimization loop
  cout << " The minimizing values are\n" << x << "\n"; //Print out the answer
}
```

It is not necessary to use the macros. They have simply been provided for convenience. If you prefer their expansion can be written out as in the following example.

```
#include <math.h>
#include <fvar.hpp>
double fcomp(dvar_vector); // Function prototype declaration
void main()
{
  int nvar=20; // This is the number of independent variables
  independent_variables x(1,nvar); // these are the independent variables
  double f;  // This is the dependent variable
  dvector g(1,nvar);  // Holds the vector of partial derivatives (the gradient)
  fmm fmc(nvar);      // Create structure to manage minimization
  // Expansion of BEGIN_MINIMIZATION(nvar,x,f,g,fmc) macro
  gradient_structure gs;
  while (fmc.ireturn >= 0)   // Begin loop for minimization
  {
    fmc.fmin(f,x,g);         // Calls the minimization routine
    if (fmc.ireturn > 0)     // Loop for evaluating the function and
    {                        // derivatives
    // End of the Expansion of BEGIN_MINIMIZATION macro
      f=fcomp(nvar,x);
    // Expansion of END_MINIMIZATION(nvar,g) macro
      gradcalc(nvar,g);
    }
  }
  // End of the Expansion of END_MINIMIZATION macro
  cout << " The minimizing values are\n" << x << "\n"; // Print out the answer
}
```

The minimization routine is controlled by the object `fmc` of type `fmm`. Different aspects of the minimization, such as the maximum number of function evaluations before stopping, convergence criteria, and printing options can be adjusted by the user through this object. See the chapter on the function minimizers for more details. The flow through the loops is controlled by the `integer fmc.ireturn`.

If you wish to use the conjugate gradient function minimizer instead of the quasi-Newton function minimizer you should declare the object `fmc` to be of type `fmmc` rather than type `fmm`. Nothing else needs to be changed.

An important feature of C++ is the ability of the language to overload arithmetic operations, ie. to extend their definition to user defined types. For example the plus operation `+` can be extended to user defined string objects so that if `string1` and `string2` are two string objects then `string1 + string2` produces a string object which contains the concatenation of `string1` and `string2`. AUTODIF uses this feature to extend the operations of addition, subtraction, multiplication, and division to various combinations of matrices, vectors, and numbers. As a result the matrices and vectors can be employed as objects in complex mathematical calculations while at the same time derivatives are obtained. This greatly simplifies and speeds up code development. Addition and multiplication of vectors and matrices appear in many engineering and statistical applications.

This example introduces two new types, the `dvector` and the `dmatrix`. These are the constant analogues of the `dvar_vector` and the `dvar_matrix`. These are "safe array" objects which contain their own size information and for which optional array bounds checking is provided. The memory for all these objects is managed by AUTODIF in a fashion which is transparent to the user. AUTODIF can combine these constant and variable objects automatically so that, for example, if `A` is a `dmatrix` and `M` is a `dvar_matrix` the expression `A*M` in the user code will produce an `dvar_matrix` which is the product of `A` and `M`. At the same time the array bounds on `A` and `M` will be checked to ensure that the operation is legal for these two objects.

To illustrate these vector and matrix operations we have taken the following example from Griewank (1988). The use of a cubic equation of state yields the Hemholtz energy of a mixed fluid of unit volume at the absolute temperature T as

$$f(x) = RT\sum_{i=1}^{n} x_i \log\Big(\frac{x_i}{1-b^T x}\Big) - \frac{x^T A x}{\sqrt{8} b^T x}\log\Big(\frac{1+(1+\sqrt{2})b^T x}{1+(1-\sqrt{2})b^T x}\Big)$$

where R is the universal gas constant and

$$0 \le x, b \in \mathbf{R}^n \qquad , \qquad A = A^T \in \mathbf{R}^{n\times n}$$

During the simulation of an oil reservoir, this function and its derivatives must be calculated at thousands of points in space and time.

In this example there are two new applications of the overloaded multiplication operator "*". $b^T x$ denotes the dot product of the n dimensional vector b with the n dimensional vector x, Ax denotes the multiplication of the n dimensional vector x by the $n \times n$ matrix A, and $x^T Ax$ is the result of the dot product of the vector x with the vector Ax. Here is the code for the function which calculates the Hemholtz energy using the overloaded operators.

```
//file: hem_func.cpp
#include <fvar.hpp>
void hemholtz_energy(int n, dmatrix A, dvector b, dvar_vector x, double& f)
{
  double R=.00005;
  double T=290;
  double root2=pow(2.,.5);
  dvariable btx = b*x;
  dvariable log_one_minus_btx = log(1-btx);
  dvariable z = R * T * ( x * (log(x)-log_one_minus_btx) );
  z -= (x * A * x)/(2*root2*btx) * log( (1+(1+root2)*btx)/(1+(1-root2)*btx) );
  f=value(z);
}
```

All the vector and matrix multiplications are carried out using the overloaded operators * + - which enable the code to be developed extremely quickly. Notice that the multiplication * can be carried out for any combination of dvector, dmatrix, dvar_vector, and dvar_matrix so long as the operation makes sense mathematically. AUTODIF can combine the constant types, double, dmatrix, and dvector, with the variable types, dvariable, dvar_vector, and dvar_matrix, in a manner which is transparent to the user.

Here is the the code which calls the function hemholtz_energy. //file:

```
hemholtz.cpp
#include <fvar.hpp>

void hemholtz_energy(int n, dmatrix A, dvector b, dvar_vector x, double& f);

#ifdef __BCPLUSPLUS__
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  long _stack = 20000;
#endif
void main()
{
  int n=30;
  double f;
  dvector g(1,n);                // this is the n dimensional gradient vector
  dvector b(1,n);                // define an n dimensional vector b
  dmatrix A(1,n,1,n);            // define an n by n matrix A
  independent_variables x(1,n);  // identify the independent variables
  // make up some values for the parameters
  // we don't know what reasonable values are, but it doesn't
  // matter for illustration of the method
  for (int i=1; i<=n; i++)
  {
    b[i]=(n/2.-i)/(n*n);   // This could be done using the ''fill'' routines
    x[i]=(1.+i)/(n*n);
    for (int j=1; j<=n; j++)
    {
      A[i][j]=i*j;
    }
  }
  {
    gradient_structure gs;     // An object of type gradient structure must
                               // be declared
    hemholtz_energy(n,A,b,x,f);// calculate the Hemholtz energy
    gradcalc(n,g);             // The gradient (partial derivatives) is
                               // calculated and put in the dvector g
    cout << "The gradient is:\n";
    cout << g << "\n";
  }
}
```

As a further example of to use AUTODIF's vector and matrix calculations consider the problem of estimating the parameters for a multivariariate normal model when the means and covariance matrix depend on a vector of common parameters Θ. Let Y be a vector of observations $(y_1, y_2, \ldots, y_n)$ which satifies the relationship

$$Y = A(\Theta)X + \epsilon$$

where X is a vector $(x_1, x_2, \ldots, x_m)$ of controls, and $A(\Theta)$ is a $n \times m$ matrix depending on a parameter vector $\Theta = (\theta_1, \ldots, \theta_p)$, and $\epsilon = (\epsilon_1, \epsilon_2, \ldots, \epsilon_n)$ is a random vector with a multivariate normal distribution with mean vector 0 and covariance matrix $\Sigma(\Theta)$.

The log-likelihood function for the parameters Θ is given by

$$-0.5\ln(\det(\Sigma(\Theta))) - 0.5(Y - A(\Theta)X)^T \Sigma(\Theta)^{-1}(Y - A(\Theta)X)$$

The maximum likelihood estimates for the parameters are found by maximizing the log-likelihood function with respect to Θ. This is a nonlinear optimization problem. The best methods for solving such a problem require the derivatives of the log-likelihood function with respect to the parameter vector Θ. Using AUTODIF's vector and matrix classes, the code for calculating the log-likelihood function and obtaining the derivatives can be written as follows. (Actually there are even more efficient methods of coding these calculations with AUTODIF.)

```
dvariable log_likelihood(dvector x,dvar_vector Y,dvar_matrix Sigma,
  dvar_matrix A)
 // It is assumed that Sigma and A have already been calculated by
 // other routines in terms of the parameters Theta
{
  dvariable f;
  dvar_vector diff=Y-A*x;
  f=-0.5*log(det(Sigma))-.5*diff*inv(Sigma)*diff;
  return f;
}
```

When this code is executed, the information necessary for calculating the derivatives is automatically generated wiuthout further intervention by the user. In addition optional array bounds checking is provided to ensure that all vector and matrix objects have the right dimensions for the operations performed.

What are ragged arrays?

Traditional programming languages such as FORTRAN which are often used for numerically intensive applications only support "flat" data structures such as matrices. The natural data structures for many applications simply do not come in this simple form. AUTODIF has extended these simple forms to ragged matrices and ragged three dimensional arrays. A ragged matrix is a two dimensional array where the rows of the array can contain a different number of elements and have different valid index ranges. A ragged three dimesnional array is made up of matrices of diferent sizes, and these matrices can themselves be ragged matrices.

Such ragged objects can be used to create many useful data structures. For example the “weights” in a feed-forward neural network can be best described by a ragged three dimensional array. This can be used to provide an extremely compact description of the neural network. See the chapter on neural networks for more details.

Since C++ and AUTODIF are extensible there is in principle no limit to the complexity of the data structures which can be developed. This means that you will never hit a “dead end” when developing an application.

Chapter 2

Getting Started

AUTODIF is designed to operate on 80286, 80386 and 80486 based microcomputers with a full 640 Kb main memory running under the MS–DOS opering system version 3.2 or higher. A hard disk is not required, however AUTODIF may generate several Mb of gradient information to be stored on an external device. A hard disk and a RAM disk are highly recommended for greatest speed. The AUTODIF libraries have been compiled with either in line code for a numeric coprocessor or with code to emulate a numeric coprocessor. A numeric coprocessor is also highly recommended for greatest speed. The optimum configuration would include a numeric coprocessor plus a hard disk and RAM disk with at least 2 Mb free memory for storing the temporary files created by AUTODIF. (Large applications may require even more disk memory.) AUTODIF appears to be compatible with most popular extended memory managers and multitasking systems.

The AUTODIF system consists of a header file `fvar.hpp` and two or more libraries. There is a safe library which provides bounds checking and initialization of arrays. This library should be used for code development. In addition there is an optimized libaray for faster execution of production programs. Chekc the `read.me` file on your AUTODIF distribution disk for the names and number of libraries supplied with your version of AUTODIF.

AUTODIF has been ported to a number of other platforms. the necessary information for each specific platform should be contained in a READ.ME file.

The simplest way to install the AUTODIF system is to place the header file in the directory where your compiler normally searches for header files and the libraries in the directory where your linker normally searches for libraries. If you choose this option, there is no need to specify additional paths for the compiler and linker to search for header files and libraries. Alternatively, you may wish to install the AUTODIF system in its own directory. In this case you will need to tell the compiler and linker where to look for the header file and libraries.

An installation program is included on the distribution disk. This program is invoked by typing `a:install` at the DOS prompt (assuming the distribution disk is in drive `a:`). You will then be asked for the drive of the source disk (in this case `a`) and the drive where you want to install AUTODIF. Next you will be asked for the directories in which to install the AUTODIF header file, libraries, and examples. If your C++ compiler is located in `c:\cc`, you might want to specify `c:` as the destination drive and `\cc\include` and `\cc\lib` for the directories to install the include file and library files respectively. If you press `Enter` for any of these directories the corresponding files will be installed in a directory called `\AUTODIF` on the destination drive.

Be sure to read the `read.me` file for last minute changes and compiler specific information.

To compile AUTODIF applications with the integrated development environment (IDE), the following options must be selected within IDE:

— Select **O**ptions **C**ompiler **C**ode Generation. Choose **L**arge memory model. (Optional: Select Pre-c**o**mpiled headers if you wish to speed up compilation.) Select Mo**r**e for Advanced Code generation. Select 8028**7** to generate direct 80287 inline code or **E**mulation to use numeric coprocessor emulation. (Optional: Select **D**ebug info in OBJs if you intend to use the debugger.)

— (Optional: If you intend to generate an overlay executable, Select **O**ptions **C**ompiler **E**ntry/Exit Code DOS **o**verlay.)

— Select **O**ptions **D**irectories. If you have not stored you AUTODIF header file and library files in the default directories, enter the path where you have stored the AUTODIF header file (`fvar.hpp`) under **I**nclude Directories and the path where you have stored the AUTODIF libraries under **L**ibrary Directories.

— Select **P**roject **O**pen to create a project file. Include your source files and the AUTODIF library that you wish to use.

To compile AUTODIF applications using a make file, the `bcc` or `bccx` command-line compilers and `tlink` or `tlinkx` linkers must be used.

You should invoke the following compiler and linker options (see `makefile` supplied with the examples):

`-I`– followed by the path of your AUTODIF include file.

`-ml`– to specify the large memory model.

`-L`– followed by the path of your AUTODIF libraries.

`-c`— to suppress linking after compilation.

`-v`— if you want to use the Borland debugger.

`-f287`— to use inline 80287 instructions.

You should invoke the `/c` and /v linker options must be invoked if your are using debugging.

The following files must be linked in the following order (see `makefile` supplied with the examples):

1. `c0l.obj startup module for the large memory model.`

2. `Your source code .obj files.`

3. `graphics.lib if you have a graphic application.`

4. `ado7.lib or ade.lib`

5. `fp87.lib for the inline floating-point library or emu.lib for the emulation library.`

6. `mathl.lib math library for large memory model.`

7. `cl.lib run-time library for large memory model.`

To compile AUTODIF applications with the integrated development environment (IDE), the following options must be selected within IDE:

— Select **C**ompile Com**p**ile Options. Select **M**emory Models L**a**rge to select the large memory model. Select Object **C**ode Inline 808**7** to generate direct inline 8087 code.

— Append the directories where you have stored the AUTODIF header file and the AUTODIF libraries to the environment strings `INCLUDE` and `LIB` as discussed in the compiler documentation.

You should invoke the following compiler and linker options (see `makefile` supplied with the examples):

`-I`— followed by the path of your AUTODIF include file.

`-ml`— to specify the large memory model.

`-c`— to suppress linking after compilation.

`-g`— if you want to use the Zortech debugger.

`-f`— if you want to generate in line 8087 code.

Many of the examples in this manual are included on the distribution disketted in a "self-expanding" archive program `examples.exe`. When this program is executed, it expands into several files. The installation program will install the examples in the directory you specify. A `makefile` is included to simplify compilation of the examples. Read the file `examples.doc` for a more complete description of the examples.

Chapter 3

The AUTODIF Classes

The AUTODIF floating point container classes have been designed to facilitate the calculation of derivatives. As in differential calculus they reflect the main dichotomy between constant objects for which no derivative calculations are required, and variable objects for which derivative information is generated. Both constant and variable objects can be further classified as single objects (a number), one dimensional arrays, two dimensional arrays, and so on. The constant and variable versions of each array type (container class) have been designed to appear as similar as possible so that identical code can be written for calculations involving either constant or variable objects.

	Constant Type	Variable Type
Single element	`double`, `float`	`dvariable`
1 dimensional array	`dvector`	`dvar_vector`
2 dimensional array	`dmatrix`	`dvar_matrix`
3 dimensional array	`d3array`	`dvar3_array`

AUTODIF also has one and two dimensional container classes for integers.

Single element	`int`, `long int`
1 dimensional array	`ivector`, `lvector`
2 dimensional array	`imatrix`, `lmatrix`

The `ivector` and `lvector` classes are arrays of integers of type `int` and `long int`. The `imatrix` and `lmatrix` classes are two dimensional arrays of integers of type `int` and `long int`.

Objects of the `dvariable` class are like numbers (variables). When they are used in calculations, derivative information is generated. The value of a `dvariable` is an object of type `double`. All the standard C arithmetic and logical functions have been defined for type `dvariable`. A dvariable `z` is created by the declaration

```
dvariable z;
```

Since the `dvariable` class is designed to appear to the user like a `double` it is not necessary to discuss many of the operations which can be performed. They appear to be simply the same as those performed on an object of type `double`. There is however an important difference between the behaviour of a `dvariable` and that of a `double` under a copy operation. A copy occurs, for example, when an object is passed as an argument to a function (so long as the function has not been declared to take a reference). Suppose that the function `func` has been declared

```
void func(dvariable)
```

so that `func` is a function which takes one argument of type `dvariable` and a portion of the users code looks like

```
dvariable v;
// ...
func(v);
```

When the function `func(v)` is called the copy initializer constructor will be called first in order to construct a copy of `v` and place it on the stack before passing control to `func`. For previous versions of AUTODIF (before version 1.05) the copy initializer performed a shallow copy – that is it passed a pointer to the component of the dvariable containing the `double`. As a result the `dvariable` in the function `func` and the `dvariable v` in the calling routine both point to the same `double` so that changes to the value of the `dvariable` in `func` will affect `v` even though `v` was passed by value. To get better compatibility with the behaviour of C code (and the behaviour of a `double`) for version 1.05 or later the `dvariable` copy initializer constructor has been modified so that it produces a distinct copy. This could cause a problem for old AUTODIF code. If a `dvariable` has been passed by value to a function and its value modified in the body of the function it used to be the case that the value of the `dvariable` in the calling function would be changed as well. This is no longer the case. Care must be taken to modify old code where necessary.

The question of distinct copies and the form of the copy initializer also occurs for container classes such as the AUTODIF vector and matrix classes. It is discussed below for the `dvar_vector` class.

There is one other operation on `dvariable`s which has no counterpart for `double`s, the value function.

```
double value(dvariable&);
```

This function returns a `double` which is the value of the `dvariable`. The main use of this function is to pass the value of a some computation for which derivative information has been calculated to some other part of the user's program so that the value can be used without creating a derivative calculation associated either with the passing of the value or its subsequent use in other parts of the program. It is not necessary to use `value` for stream output of `dvariable`s because the stream operators `<<` and `>>` have been overloaded to accept these objects. The value function should be used sparingly. In particular it must not be used in the middle of calculations for which the derivatives are desired. The following example will produce an error

```
dvariable x,y,z;
double a;
// ...            This code is incorrect
y=x*x;
a=value(y);
cout << " y = " << a << "\n";
z=sin(a); // This is where an error is introduced into the derivatives
cout << " z = " << z << "\n"; This will print out the value of z
```

The following correct version of the code will produce exactly the same output and the derivative calculations will be correct

```
dvariable x,y,z;
double a;
// ...            This code is correct
y=x*x;
cout << " y = " << y << "\n";
z=sin(y); // Use y instead of a at this point
cout << " z = " << z << "\n"; This will print out the value of z
```

It is possible to create an array `u` of 10 `dvariables` by using the declaration

```
dvariable u[10]; // creates an array of 10 dvariables indexed from 0..9
```

However, this is not the recommended construction. Instead the `dvar_vector` class should be used as in the declaration

```
dvar_vector u(0,9); // creates a dvar_vector of size 10 indexed from 0..9
```

This construction generates more efficient code and produces a “safe” object with optional bounds checking. The `dvariable u[10]` construction may be useful for converting preexisting C code where the declaration `double` or `float` is modified to `dvariable`.

There is at present no AUTODIF type which corresponds to the C type `float`. All AUTODIF variable types contain a `double`.

Since the `dvector` and `dvar_vector` classes are intended to appear identical to the user, with the exception that derivative information is generated for the `dvar_vector` class, we shall discuss the use of these classes simultaneously.

The `dvar_vector` class appears to the user as a vector (array) of objects of type `dvariable`. Bounds checking for array and matrix objects can be implemented or not by linking the user's program with the appropriate AUTODIF library. The index for a `dvector` a `dvar_vector` is of type `int` (a signed integer) so that permissible array bounds are -32,768 to 32,767 for 16 bit DOS versions where an object of type `int` occupies two bytes. The maximum number of elements which can be contained in a `dvar_vector` is 8191 in 16 bit DOS versions of AUTODIF.

Several declarations can be used to create a `dvector` or `dvar_vector` object.

```
dvector v(1,n); // Creates a dvector with array bounds 1..n
dvector v=w;    // Creates a dvector and initializes it with w
dvector w("{1.2,4,-5.23e-1}");  // Creates a dvector with array bounds 1..3
                     // and initializes it with 1.2, 4, -5.23e-1.
dvector w("mystuff.dat"); //Creates a dvector whose contents are
                     // read in from the file mystuff.dat. The data in
                     // mystuff.dat must be invalid numeric format
```

The code `dvector v=w;` creates a `dvector v` by using the "copy initializer" `dvector(dvector&)` constructor. This constructor performs a shallow copy so that after this line of code is executed `v` and `w` will both point to the same area of memory where the array is contained. The same type of constructor is invoked when a `dvector` or `dvar_vector` object is passed by value to a function. For example suppose that a function `users_function` has been defined where the function prototype is

```
void users_function(dvar_vector v);
```

and that the following statement appears somewhere in the user's code.

```
dvar_vector w(1,n);
. . .
. . .
users_function(w);
. . .
```

To pass the `dvar_vector w` to `users_function`, a copy of `w` is made and put on the stack. The copy initializer constructor is called by the compiler to do this. The copy initializer uses a "shallow" copy to create the copy of `w` which is passed to the function.

A `dvector` contains a pointer which points to the area of memory where the elements of the `dvector` are contained. With a shallow copy the value of this pointer is simply passed to the new copy of the `dvector w`. As a result, both `w` in the calling routine and the copy of `w` in the function `users_function` point to the same area of memory so that changes made to this `dvector` in `users_function` will change the values of the corresponding entries of `w` in the calling routine even thought the `dvector` was passed by value. This can create a problem if the user wants to operate on the `dvector` in `users_function` without changing the value of `w` in the calling routine.

For ordinary C language objects, changes within a function to an argument passed by value to the function do not affect the object passed. This is not the case for the AUTODIF container classes `dvariable, dvector, dvar_vector, dmatrix, or dvar_matrix` or higher dimensional arrays. If you wish to pass a "copy" of an object to a function so that the original object is not affected by changes to the passed object you must explicitly make a distinct copy of the object by using the assignment operator `=` as in this example. .

```
dvar_vector v(1,n);  // want to pass a copy of v to the function func
dvar_vector w(1,n);  // This will be the distinct copy of v
w=v;                 // This creates the distinct copy
users_function(w);             // Can now call the function
```

The user must avoid the simpler C++ construction of declaring an initialized object. Such a declaration will cause the copy initializer to be invoked so that a shallow copy will be performed.

```
dvar_vector v(1,n);
dvar_vector w=v;     // this uses the copy initializer so that w is
                     // a shallow copy of v
users_function(w);   // Changes to w in the function will cause changes
                     // in v
```

It is possible to create a dvector by explicitly naming the data it will contain

```
dvector w("{12,24,56 78 91.4 23.455e+2}");
```

The data must be enclosed in double quotes and begin and end with braces { }. The numbers must be in valid numeric format and be separated by blanks or commas. It is also possible to create a dvector using the contents of a file. The declaration

```
dvector w("mystuff.dat");
```

will create a `dvector w` which contains the contents of the file `mystuff.dat`. It is not necessary to know how may numbers are contained in the file to use this declaration. An error message will be generated if the file does not exist, the file is empty, or there is non-numeric data in the file. The data in the file must be separated by spaces. This constructor will create a `dvector` whose minimum valid index is 1.

It is also possible to create a `dvector` or a `dvar_vector` by reading in a column of data from a file. Suppose the file ‘‘`work.prn`’’ contains the data

Monday	25.12	-41	433.12	10.11	23
Tuesday	34.21	356	23	23.1	5
Wednesday	10	3.13e-4	43.11	3.23	4
Thursday	12.1	53.13	453.1	-5.13	4.1

The declaration

```
dvar_vector u("work.prn",3);
```

will create a `dvar_vector` whose elements are the third column of the file `work.prn`. That is

```
u = (-41,356,3.13e-4,53.13)
```

The entries of the file must be separated by blanks and all the entries in the column being read must be in valid numeric format. This constructor will create a `dvector` whose minimum valid index is 1.

Since this version of AUTODIF does not feature resizeable arrays, the only way that you can place the producet of two `dvar_matrices M1` and `M2` in a `dvar_matrix M3` without knowing or calculating the size of the product is to use the construction

```
dvar_matrix M3=M1*M2;
```

so that the copy initializer constructor will create an object with the correct size. Of course it is possible to calulate the size of the product and explicitly declare the `dvar_vector M3` by using the `dvar_matrix` access functions to obtain the row and column dimensions of the matrices being multiplied as in the code

```
dvar_matrix M3(M1.rowmin(),M1.rowmax(),M2.colmin(),M2.colmax());
M3=M1*M2;
```

but this solution lacks elegance.

Most of the components of the AUTODIF classes are private so that they can not be accessed directly by the user. To allow the user to access the relevant elements of the classes, public class member access functions are provided.

```
int indexmin();     //Returns the minimum allowable index  for an array
int indexmax();     //Returns the maximum allowable index  for an array
int size();         //Returns the size of an array
int shift(int min); //Changes the allowable indices to min and min+size-1
```

These functions can be used to determine the size and valid index ranges of a vector object which has been passed to a user's function. Thus it is not necessary to explicitly pass such information when designing a function which uses a vector object. The information is already passed with the object itself. If `v` is a `dvector`, the function `v.indexmin()` will return the minimum valid index for `v`. The `shift` function allows the valid index range for a `dvector` or `dvar_vector` to be changed. As an example of a function which acts on a vector objects, here is a listing for a function which overloads the multiplication operator `*` so that `v * w` will be the dot product (also sometimes called the scalar product) of the `dvectors v` and `w`.

```
double operator * (dvector& t1, dvector& t2)
{
  // Check to see if vectors have the same valid index bounds
  if (t1.indexmin() != t2.indexmin() ||  t1.indexmax() != t2.indexmax())
  {
    cerr << "Index bounds do not match in double operator * "
            "(dvector&,dvector&)\n";
    exit(1);  // Stop if vectors are not compatible
  }
  double tmp;
  tmp=0;
  for (int i=t1.indexmin(); i<=t1.indexmax(); i++)
  {
    tmp+=t1[i]*t2[i];
  }
  return(tmp);
}
```

If you write your own functions it is a good idea to provide both a constant and variable version at the time of writing. This is especially true if you write a variable version of the routine. Suppose that the user has written a function `void userfun(dvar_matrix)` which takes a `dvar_matrix` argument, but has neglected to make a version `void userfun(dmatrix)` which takes a `dmatrix` argument. At some later time the user wants to use the function `userfun` with an argument of type `dmatrix` such as in the following code segment.

```
{
  dmatrix M(1,5,1,5);
  userfun(M); // M will be converted to a dvar_matrix
  // ...

}
```

Since there is no version of `userfun` which accepts an argument of type `dmatrix` the C++ compiler will convert `M` to an object of type `dvar_matrix` by invoking the `dvar_matrix(dmatrix&)` constructor. This is probably not what the user intends and will be disasterous if an object of type `gradient_structure` is not in scope at the time.

The corresponding variable version of the `dot` product for `dvar_vectors` would look like

```
dvariable operator * (dvar_vector& t1, dvar_vector& t2)
{
  if (t1.indexmin() != t2.indexmin() ||  t1.indexmax() != t2.indexmax())
  {
    cerr << "Index bounds do not match in dvariable operator * "
            "(dvar_vector&,dvar_vector&)\n";
    exit(1);
  }
  RETURN_ARRAYS_INCREMENT();
  dvariable tmp;
  tmp=0;

  for (int i=t1.indexmin(); i<=t1.indexmax(); i++)
  {
    tmp+=t1[i]*t2[i];
  }
  RETURN_ARRAYS_DECREMENT();
  return(tmp);
}
```

The appearance of the statements `RETURN_ARRAYS_INCREMENT();` and `RETURN_ARRAYS_DECREMENT();` requires some explanation. AUTODIF employs a stack of queues to hold intermediate arithmetic results for variable objects. To ensure that these queues are not overflowed you should always increment the queue structure when entering a function which returns a variable object and decrement the queue structure when leaving the function. If you fail to increment the queue structure and a queue overflow occurs the program will continue to operate, but the arithmetic will be corrupted. If you fail to decrement the queue structure when leaving the function you may eventually increment past the end of the stack of queues producing an error message

```
Overflow in RETURN_ARRAYS stack -- Increase NUM_RETURN_ARRAYS
There may be a RETURN_ARRAYS_INCREMENT()
which is not matched by a RETURN_ARRAYS_DECREMENT()
```

Simply include the increment and decrement statements in every function you create which returns a variable object and everything will be fine. It is not necessary to include the increment and decrement statements in a function which does not return a variable type because such a function can not be used in an arithmetic statement. **You should never include the `RETURN_ARRAYS_INCREMENT()` and `RETURN_ARRAYS_DECREMENT()` functions in a function which returns a void or constant type such as `double`, `dvector`, or `dmatrix`.** The reason is that such a function might be called when there is no object from the class `gradient_structure` in scope. Calling `RETURN_ARRAYS_INCREMENT()` or `RETURN_ARRAYS_DECREMENT()` when there is no object of type gradient_structure in scope could cause a serious program error.

Both the `()` and the `[]` operators can be used to access elements of arrays. There is absolutely no difference between them. If `v` is a `dvector` the i'th element of `v` can be accessed by writing either `v[i]` or `v(i)`. Optional array bounds checking is provided for both operators. Array bounds checking is implemented or turned off by linking with the appropriate library. No changes in the user's code are required. Linking with the safe libraries enables array bounds checking, while linking with the optimized libraries disables array bounds checking.

It is important to recognize that there is a difference in AUTODIF between a vector object and a matrix object which has either 1 row or 1 column. This means that a vector object is neither a row vector nor a column vector, it is simply a vector object. We have adopted this treatment of vector objects so that it is possible to multiply vector objects such as `u` and `v` simply by writing `u*v`. If vector objects were the same as matrices with 1 column, it would be necessary to write `trans(u)*v` to multiply two vectors. Here `trans` denotes the transpose of a `matrix` object.

Sometimes it is useful to be able to treat vector objects as though they were matrices with one row or column. The functions `column_vector` and `row_vector` enable this to be done. The function

```
dmatrix column_vector(dvector& v);
```

will convert v into a matrix with 1 column. The valid column index is from 1 to 1. The valid row index bounds are the same as the valid index bounds for `v`. The function

```
dmatrix row_vector(dvector& v);
```

will convert v into a matrix with 1 row. The valid row index is from 1 to 1. The valid column index bounds are the same as the valid index bounds for `v`.

As an example of how to use these function consider the problem of computing the outer product of two vectors. The outer product of two vectors (u_i) and (v_j) is the matrix w_{ij} where $w_{ij} = u_i v_j$. The statement

```
dmatrix w=column_matrix(v)*row_matrix(u);
```

will compute the outer product of the `dvectors u` and `v`. Of course you can also use the function `outer_prod` supplied with the AUTODIF libraries to accomplish the same thing.

```
dmatrix w=outer_prod(u,v);
```

Another use for the `column_vector` function is to print out a vector one element to a line. The overloaded operator `<<` will write out a vector object all on one line. This is useful if you want to put a vector object into a row in a spreadsheet, for example, but it can be a problem for some editors which can not read in such long lines in a file. The code

```
cout << column_vector(v);
```

will print out the vector v with one entry per line.

The `dmatrix` and `dvar_matrix` classes are implemented as arrays of `dvector` and `dvar_vectors`. If `M` is a `dmatrix` then `M[j]` or `M(j)` is a `dvector` whose elements are the `j`th row of `M`. Since each row is a `dvector` which contains its own size (shape) information it is a simple matter to construct `dmatrix` objects which have a different numbers of rows in each column, so called "ragged" matrices. Of course these are not matrices in the usual sense in which the word is used, but they can be useful container objects for some applications so we have extended the `dmatrix` and `dvar_matrix` classes to include them. As with the `dvector and dvar_vector` classes `dmatrix and dvar_matrix` classes appear identical to the user with the exception that derivative information is collected for the `dvar_matrix class`. An object of type `dvar_matrix` is created by using the declarations

```
dmatrix M(lbr,lur,lbc,luc); // Creates a (lur-lbr+1)×(luc-lbc+1) matrix
dmatrix M1=M;               // M has already been defined. M1 is initialized
                            // by M.
dmatrix M(lbr,lur,ivec_lb,ivec_ub); // Creates a ''ragged'' dmatrix with a
                            // variable number of rows in each column
dmatrix M("filename");      // Creates a dmatrix whose elements are the
                            // contents of the file ''filename''
dmatrix M( "{ 1.5,-2.0,3.0,3.5 }"
         "{ 2.0,14.0,5.1,2.2 }" );// creates and initializes a dmatrix
```

The constructor `dmatrix(lbr,lur,ivec_lb,ivec_ub)` creates a `dmatrix` with `lur-lbr+1` rows. The `i`th row is a `dvector` of size `ivec_ub[i]-ivec_lb[i]+1` and having a range of valid indices running from `ivec_lb[i]` to `ivec_ub[i]`. `ivec_lb` and `ivec_ub` are instances of the class `ivector`, that is vectors of integers. They must have valid indices ranging from `lbr` to `lur`. For example suppose we wish to make a ragged `dvar_matrix` with 4 rows for which the valid row index bounds should be from 2 to 5. The declaration should be

```
dvar_matrix M(2,5,ivec_lb,ivec_ub);
```

Suppose the desired valid row indices are 1 to 6 for row 1, 0 to 3 for row 2, 1 to 2 for row 4, and 3 to 9 for row 5. The legal bounds on the ivectors ivec_lb and ivec_ub must be equal to the legal column bounds on M. We can create such ivector objects with the declaration

```
ivector ivec_lb(2,5);
ivector ivec_ub(2,5);
```

The correct values for ivec_lb and ivec_ub can be inserted for example by using the “fill” function

```
ivec_lb.fill("{1,0,1,3}");
ivec_ub.fill("{6,3,2,9}");
```

Alternatively the desired ivector objects can be created with the declarations

```
ivector ivec_lb("{1,0,1,3}");
ivector ivec_ub("{6,3,2,9}");
```

These declarations will create ivector objects whose legal bounds go from 1 to 4. To change the valid index bounds to the desired values from 2 to 5 the class member function shift() can be used.

```
ivec_lb.shift(2); // legal bounds will be from 2 to 5
ivec_ub.shift(2); // legal bounds will be from 2 to 5
```

As is the case for vector objects, the copy initializer for matrix objects performs a light copy. To create a distinct new matrix object you must use the assignment operator =.

The declaration

```
dmatrix M("filename");
```

will create a dmatrix M whose elements are the contents of the file filename. All entries in the file must be in valid numeric format. All entries must be separated by blanks or commas. Each line in the file becomes a line of the matrix. Ragged matrices are supported so that the files may have different number of entries in each line. For the DOS version of AUTODIF no line may contain more than 6550 elements.

To create a dmatrix and initialize it with data a declaration of the form

```
dmatrix W( "{ 1.5,-2.0,3.0,3.5 }"
           "{ 2.0,14.0,5.1,2.2 }" );
```

can be used. This declaration will create a dmatrix with 2 rows and 4 columns. The minimum valid index for the rows and columns is set equal to 1.

The components of all the AUTODIF classes are private which means that they can not be accessed directly by the user. Public class member access functions are provided to allow the user to access the relevant elements of the classes.

```
int rowmin();   //Returns the minimum allowable row index  for a matrix
int rowmax();   //Returns the maximum allowable row index  for a matrix
int colmin();   //Returns the minimum allowable column index  for a matrix
int colmax();   //Returns the maximum allowable column index  for a matrix
int rowsize();  //Returns the number of rows in an array
int colsize();  //Returns the number of columns in an array
void colshift(int&);  //Changes the range of valid indices for the columns
void rowshift(int&);  //Changes the range of valid indices for the rows
```

Three dimensional arrays have been implemented in this version of AUTODIF mainly as containers for collections of two dimensional arrays. If `u` is a `d3array` then `u[1]` or `u(1)` is a dmatrix (provided that 1 is a valid index for `u`). We shall refer to the object obtained by fixing the first index of a three dimensional array as a "slice". An element of a three dimensional array is determined by picking a slice, a row, and a column.

An object of type `dvar3_array` is created by using the declarations

```
dvar_3darray M(ls,us,lr,ur,lc,uc); // Creates a (us-ls+1)*(ur-lr+1)*(uc-lc+1)
                                   // dvar_3darray
dvar_3darray M1=M;                 // M has already been defined. M1 is
                                   // initialized by M.
d3array(ls,us,ivec_lr,ivec_ur,ivec_lc,ivec_uc); // Creates a ragged d3array
                            // with a variable size dmatrices in each slice.
```

```
int slicemin();   //Returns the minimum allowable slice index  for a d3array
int slicemax();   //Returns the maximum allowable slice index  for a d3array
int rowmin();   //Returns the minimum allowable row index  for a d3array
int rowmax();   //Returns the maximum allowable row index  for a d3array
int colmin();   //Returns the minimum allowable column index  for a d3array
int colmax();   //Returns the maximum allowable column index  for a d3array
int slicesize();  //Returns the number of slices in a d3array
int rowsize();  //Returns the number of rows in an d3array
int colsize();  //Returns the number of columns in an array
void sliceshift(int&);  //Changes the range of valid indices for the slices
void colshift(int&);  //Changes the range of valid indices for the columns
void rowshift(int&);  //Changes the range of valid indices for the rows
```

It is often useful to be able to convert a floating point object into an integer object or an integer object into a floating point object. (See for instance the example on creating a bootstrap sample where a `dvector` object of random numbers is converted into an `ivector` object of integer indices). It is possible to convert between vector objects as follows:

```
ivector <--> dvector
lvector <--> dvector
lvector <--> ivector
```

These conversion may be made explicit as in the following code fragment

```
dvector u(1,10);
ivector iv(1,10);
// ...
iv = ivector(u); // iv will be filled with the integer parts of the components
                 // of u
```

or the conversion may be "implicit", that is, supplied by the compiler.

```
dvector u(1,10);
// ...
ivector iv = u; // iv will be filled with the integer parts of the components
                // of u
```

In the second example the `ivector iv` will have minimum and maximum valid indices determined by the minimum and maximum valid indices of the `dvector u`.

Chapter 4

Operations and Functions

Since AUTODIF supplies a large number of operators and functions there is often more than one way to carry out a given calculation. While all equivalent methods will produce the same answer, one method may produce more efficient code. For example, the two lines of code

```
A=A+B;
//
A+=B;
```

will both add the object `A` to the object `B` and store the result in A. (This will work whether `A` and `B` are numbers, vectors , or matrices.) However the code produced by `A+=B` is more efficient in all cases. It is both faster and generates less temporary data to be stored.

While the type of object produced by an arithmetic operation depends on whether the arguments are constants (`double,dvector or dmatrix`) or variables (`dvariable, dvar_vector, or dvar_matrix`) the sort of operation defined by the operation does not. The symbol `+` between two vector objects will denote the component-wise sum of these vectors and produce a vector object corresponding to their sum. This is true regardless of whether one or both objects are constant or variable objects. We denote this by writing

```
vector_object = vector_object + vector_object // vector sum
```

We shall refer to the "output" object on the left side of the "=" sign by z if it is a number, z_i if it is a vector and z_{ij} if it is a matrix in order to express the components of the operation. We shall refer to the first input object (the one on the right side of the "=") by x, x_i, and x_{ij} and the second input object if there is one by y, y_i, and y_{ij} For any operation the property of being a variable is dominant in that if either of the input objects is a variable then the output object is a variable.

```
vector_object = vector_object + vector_object // vector sum
```

$$z_i = x_i + y_i$$

```
vector_object = number + vector_object // add number to vector
```

$$z_i = x + y_i$$

```
vector_object = vector_object + number // add number to vector
```

$$z_i = x_i + y$$

```
matrix_object = matrix_object + matrix_object // matrix sum
```

$$z_{ij} = x_{ij} + y_{ij}$$

```
matrix_object = matrix_object + number // matrix sum
```

$$z_{ij} = x_{ij} + y$$

```
vector_object = vector_object - vector_object // vector difference
```

$$z_i = z_i - y_i$$

```
vector_object = vector_object - number // subtract number from a vector
```

$$z_i = x_i - y$$

```
matrix_object = matrix_object - number // matrix difference
```

$$z_{ij} = x_{ij} - y$$

```
matrix_object = matrix_object - matrix_object // matrix difference
```

$$z_{ij} = x_{ij} - y_{ij}$$

```
number = vector_object * vector_object // vector dot product
```

$$z = \sum_i x_i y_i$$

```
vector_object = number * vector_object // scalar product
```

$$z_i = x * y_i$$

```
vector_object = vector_object * number // scalar product
```

$$z_i = x_i * y$$

```
vector_object = vector_object * matrix_object // vector times matrix
```

$$z_j = \sum_i x_i * y_{ij}$$

```
vector_object = matrix_object * vector_object // matrix times vector
```

$$z_i = \sum_j x_{ij} * y_j$$

```
matrix_object = matrix_object * matrix_object // matrix times matrix
```

$$z_{ij} = \sum_k x_{ik} * y_{kj}$$

```
matrix_object = number * matrix_object // scalar times matrix
```

$$z_{ij} = x * y_{ij}$$

An additional vector–matrix operation is the outer product of two vectors.

```
matrix_object = outer_prod(vector_object,vector_object) // outer product of two vectors
```

$$z_{ij} = x_i * y_j$$

These operators generate faster code than the corresponding operators defined above since their use avoids the necessity for creating a temporary object to hold the result and the extra overhead associated with assigning the values contained in this temporary object to the left hand side of the expression.

```
vector_object += vector_object // vector sum
```

$$x_i+=y_i$$

```
vector_object += number // add number to vector
```

$$x_i+=d$$

```
vector_object -= vector_object // vector difference
```

$$x_i-=y_i$$

```
vector_object -= number // subtract number from vector
```

$$x_i-=d$$

```
vector_object /= number // divide vector by number
```

$$x_i/=d$$

```
vector_object *= number // multiply vector by number
```

$$x_i*=d$$

```
matrix_object += matrix_object // matrix sum
```

$$x_{ij}+=y_{ij}$$

```
matrix_object += number // add number to matrix
```

$$x_{ij}+=d$$

```
matrix_object -= matrix_object // matrix subtraction
```

$$x_{ij}-=y_{ij}$$

```
matrix_object -= number // subtract number from matrix
```

$$x_{ij}-=d$$

```
vector_object = vector_object / number // divide a vector by a number
```

$z_i = x_i/y$

```
vector_object = number / vector_object // divide a vector by a number
```

$z_i = x/y_i$

```
matrix_object = matrix_object / number // divide a matrix by a number
```

$z_{ij} = x_{ij}/y$

The & operator has been overloaded to produce the concatenation of two vector objects.

```
vector_object=vector_object & vector_object
```

If x=$(x_r, \ldots, x_n)$ and y=$(y_s, \ldots, y_m)$ then x & y= $(x_r, \ldots, x_n, y_s, \ldots, y_m)$ and the minimum valid index for x & y is equal to the minimum valid index for x.

Since in the present version of AUTODIF arrays are not resizeable it is not possible to write something like

```
x = x & y;
```

where x and y are dvectors. Instead you must do something like

```
dvector z = x & y;
```

Future version of AUTODIF will incorporate resizeable arrays.

There are several operations familiar to users of spreadsheets which do not appear as often in classical mathematical calculations. For example spreadsheet users often wish to multiply one column in a spreadsheet by the corresponding elements of another column. Spread sheet users might find it much more natural to define the product of matrices as an element-wise operation such as

$z_{ij} = x_{ij} * y_{ij}$

The “classical” mathematical definition for the matrix product has been assigned to the overloaded operator “*” so that large mathematical formulas involving vector and matrix operations can be written in a concise notation. Typically, spreadsheet-type calculations are not so complicated and do not suffer so much from being forced to adopt a “function-style” of notation.

Since addition and subtraction are already defined in an element-wise manner, it is only necessary to define element-wise operations for multiplication and division. We have name these functions `elem_prod` and `elem_div`.

```
vector_object = elem_prod(vector_object,vector_object) // element-wise multiply
```

$z_i = x_i * y_i$

```
vector_object = elem_div(vector_object,vector_object) // element-wise divide
```

$z_i = x_i / y_i$

```
matrix_object = elem_prod(matrix_object,matrix_object) // element-wise multiply
```

$z_{ij} = x_{ij} * y_{ij}$

```
matrix_object = elem_div(matrix_object,matrix_object) // element-wise divide
```

$z_{ij} = x_{ij} / y_{ij}$

```
matrix_object = identity_matrix(int min,int max)
```

Creates a square identity matrix with minimum valid indices min and maximum valid index max.

The determinant of a matrix object (The matrix must be square, that is the number of row must equal the number of columns)

```
matrix_object = det(matrix_object)
```

The inverse of a matrix object (The matrix must be square, that is the number of row must equal the number of columns)

```
matrix_object = inv(matrix_object)
```

The norm of a vector_object

```
number = norm(vector_object)
```

$$z = \sqrt{\sum_i x_i^2}$$

The norm squared of a vector_object

```
number = norm2(vector_object)
```

$$z = \sum_i x_i^2$$

The norm of a matrix_object

```
number = norm(matrix_object)
```

$$z = \sqrt{\sum_{ij} x_{ij}^2}$$

The norm squared of a matrix_object

```
number = norm2(matrix_object)
```

$$z = \sum_{ij} x_{ij}^2$$

The sum over the elements of a vector object

```
number = sum(vector_object)
```

$$z = \sum_i x_i$$

The row sums of a matrix object

```
vector = rowsum(matrix_object)
```

$$z_i = \sum_j x_{ij}$$

The column sums of a matrix object

```
vector = colsum(matrix_object)
```

$$z_j = \sum_i x_{ij}$$

The minimum element of a vector object

```
number = min(vector_object)
```

The maximum element of a vector object

```
number = max(vector_object)
```

While we have included eigenvalue and eigenvector routines for both constant and variable matrix objects you should be aware that in general the eigenvectors and eigenvalues are not differentiable functions of the variables determining the matrix.

The eigenvalues of a symmetric matrix

```
vector_object = eigenvalues(matrix_object)
```

are returned in a vector. It is the users responsibility to ensure that the matrix is actually symmetric. The routine symmetrizes the matrix so that the eigenvalues returned are actually those for the symmetrized matrix.

The eigenvectors of a symmetric matrix

```
matrix_object = eigenvectors(matrix_object)
```

are returned in a matrix. It is the users responsibility to ensure that the matrix is actually symmetric. The routine symmetrizes the matrix so that the eigenvectors returned are actually those for the symmetrized matrix. The eigenvectors are located in the columns of the matrix. The i'th eigenvalue returned by the function `eigenvalues` corresponds to the i'th eigenvector returned by the function `eigenvector`.

For a positive definite symmetric matrix `S`, the choleski decomposition of `S` ia a lower triangular matrix `T` satifying the relationship `S=T*trans(T)`. If `S` is a (positive definite symmetric) matrix object and `T` is a matrix object, the line of code

```
T=choleski_decomp(S);
```

will calculate the choleski decomposition of `S` and put it into `T`.

If `y` is a vector and `M` is an invertible matrix then finding a vector `x` such that

```
x=inv(M)*y
```

will be referred to as solving the system of linear equations determined by `y` and `M`. Of course it is possible to use the `inv` function to accomplish this task but it is much more efficient to use the `solve` function.

```
vector x=solve(M,y); // x will satisfy x=inv(M)*y;
```

It turns out that it is a simple matter to calculate the determinant of the matrix `M` at the same time as the system of linear equations is solved, and since this is useful in multivariate analysis we have also included a function which returns the determinant at the same time as the system of equations is solved. To avoid floating point overflow or underflow when working with large matrices the logarithm of the absolute value of the determinant together with the sign of the determinant are returned The constant form of the solve function is

```
double ln_det;
double sign;
dvector x=solve(M,y,ln_det,sign);
```

while the variable form is

```
dvariable ln_det;
dvariable sign;
dvar_vector x=solve(M,y,ln_det,sign);
```

The solve function is useful for calculating the log-likelihood function for a multivariate normal distribution. Such a log-likelihood function involves a calculation similar to

```
  l = -.5*log(det(S)) -.5*y*inv(S)*y
```

where Sis a matrix object and y is a vector object. It is much more efficient to carry out this calculation using the solve function. The following code illustrates the calculations for variable objects.

```
   dvariable ln_det;
   dvariable sign;
   dvariable l;
   dvar_vector tmp=solve(M,y,ln_det,sign);
   l=-.5*ln_det-y*tmp;
```

While it is always possible to fill vectors and matrices by using loops and filling them element by element, this is tedious and prone to error. To simplify this task a selection of methods for filling vectors and matrices with random numbers or a specified sequence of numbers is available. There are also methods for filling row and columns of matrices with vectors. In this section the symbol `vector` can refer to either a `dvector` or a `dvar_vector`. while the symbol `matrix` can refer to either a `dmatrix` or a `dvar_matrix`.

```
  void vector::fill("{m,n,...,}")
```

fills a vector with a sequence of the form `n, m,` ... The number of elements in the string must match the size of the vector.

```
  void vector::fill_seqadd(double& base, double& offset)
```

fills a vector with a sequence of the form `base, base+offset, base+2*offset,`...

For example if `v` is a `dvector` created by the statement

```
dvector v(0,4);
```

then the statement

```
v.fill_seqadd(-1,.5);
```

will fill v with the numbers $(-1.0, -0.5, 0.0, 0.5, 1.0)$.

```
void matrix::rowfill_seqadd(int& i,double& base, double& offset)
```

fills row i of a matrix with a sequence of the form base, base+offset, base+2*offset,...

```
void matrix::colfill_seqadd(int& j,double& base, double& offset)
```

fills column j of a matrix with a sequence of the form base, base+offset, base+2*offset,...

```
void matrix::colfill(int& j,vector&)
```

fills the j'th column of a matrix with a vector

```
void matrix::rowfill(int& i,vector&)
```

fills the i'th row of a matrix with a vector

In this section a uniformly distributed random number is assumed to have a uniform distribution on $[0, 1]$. A normally distributed random number is assumed to have mean 0 and variance 1. A binomially distributed random number is assumed to have a parameter p where 1 is returned with probability p and 0 is returned with probability $1 - p$. A multinomially distributed random variable is assumed to have a vector of parameters P where i is returned with probability p_i. If the components of P do not sum to 1 the vector will be normalized so that the components do sum to 1.

```
void vector::fill_randu(long int& n)
```

fills a vector with a sequence of uniformly distributed random numbers. The long int n is a seed for the random number generator. Changing n will produce a different sequence of random numbers.

```
void matrix::colfill_randu(int& j,long int& n)
```

fills column j of a matrix with a sequence of uniformly distributed random numbers The long int n is a seed for the random number generator. Changing n will produce a different sequence of random numbers.

```
void matrix::rowfill_randu(int& i,long int& n)
```

fills row i of a matrix with a sequence of uniformly distributed random numbers

```
void vector::fill_randbi(long int& n, double& p)
```

fills a vector with a sequence random numbers from a binomial distribution.

```
void vector::fill_randn(long int& n)
```

fills a vector with a sequence of normally distributed random numbers

```
void matrix::colfill_randn(int& j,long int& n)
```

fills column j of a matrix with a sequence of normally distributed random numbers

```
void matrix::rowfill_randn(int& i,long int& n)
```

fills row i of a matrix with a sequence of normally distributed random numbers

```
void vector::fill_multinomial(long int& n, dvector& p)
```

fills a vector with a sequence random numbers from a multinomial distribution. The parameter p is a dvector such that p[i] is the probability of returning i. The elements of p must sum to 1.

```
vector extract_column(matrix& M,int& j)
```

extracts a column from a matrix and puts it into a vector

```
vector extract_row(matrix& M,int& i)
```

extracts a row from a matrix and puts it into a vector.

```
vector extract_diagonal(matrix& M)
```

extracts the diagonal elements from a matrix and puts them into a vector.

The function call operator ( ) has been overloaded inv two ways to provide for the extraction of a subvector.

```
vector(ivector&)
```

An ivector object is used to specify the elements of the vector to be chosen. If u and v are dvectors and i is an ivector the construction

```
dvector u = v(i);
```

will extract the members of v indexed by i and put them in the dvector u. The size of u is equal to the size of i. The dvector u will have minimum valid index and maximum valid index equal to the minimum valid index and maximum valid index of i. The size of i can be larger than the size of v in which case some elements of v must be repeated. The elements of the ivector i must lie in the valid index range for v.

If v is a dvector and i1 and i2 are two integers

```
u(i1,i2)
```

is a dvector which is a subvector of v (provided of course that i1 and i2 are valid indices for v). Subvectors can appear on both the left and right hand side of an assignment.

```
dvector  u(1,20);
dvector v(1,19);
v = 2.0;   // assigns the value 2 to all elements of v
u(1,19) = v; // assigns the value 2 to elements 1 through 19 of u
```

In the above example suppose that we wanted to assign the vector v to elements 2 through 20 of the vector u. To do this we must first ensure that they have the same valid index ranges. The operators ++ and -- increment and decrement the index ranges by 1. The code fragment

```
dvector  u(1,20);
dvector v(1,19);
v = 2.0;   // assigns the value 2 to all elements of v
--u(2,20) = v; // assigns the value 2 to elements 2 through 20 of u
u(2,20) = ++v; // assigns the value 2 to elements 2 through 20 of u
```

It is important to realize that from the point of view of the vector v both of the above assigments have the same effect. It will have elements 2 through 20 set equal to 2. The difference is in the side effects on the vector v. The use of subvectors and increment and decrement operations can be used to remove loops from the code. Note that

```
dvector x(1,n)
dvector y(1,n)
dvector z(1,n)
for (int i=2;i<=n;i++)
{
  x(i)=y(i-1)*z(i-1);
}
```

can be written as

```
dvector x(1,n)
dvector y(1,n)
dvector z(1,n)
x(2,n)=++elem_prod(y(1,n-1),z(1,n-1));  // elem_prod is element-wise
                                        // multiplication of vectors
```

As an example of how one can use some of these operations, consider the problem of creating bootstrap samples from a set of data. We wish to create a random sample (with replacement) of the original data set. Assume that the original data is contained in a dvector object named data. It is not necessary that the new samples be the same size as the original sample so we shall allow the size of the bootstrap sample to be a variable. The code for the function bootstrap follows.

```
dvector bootstrap(dvector& data, int& sample_size, long int seed)
{
  // data contains the data which is used to make the bootstrap sample
  // sample_size is the size of the desired bootstrap sample
  // seed is a seed for the random number generator
  dvector tmp(1,sample_size);
  tmp.fill_randu(seed);  // 0<tmp(j)<1             for 1<j<sample_size
  tmp=tmp*data.size()+1; // 1<tmp(j)<data.size()+1 for 1<j<sample_size
  ivector iselect(tmp);  // Take the integer parts of tmp and put them
                         // into iselect
                         // 1<=iselect(j)<=data.size() for 1<=j<=sample_size
  tmp=data(iselect);     // Select the bootstrap sample and
                         // store it in tmp
  return(tmp);
}
```

The above code could be employed in the user's code to make bootstrap samples by using something like the following code fragment

```
// ...
dvector data(1,nobs)
// ... somehow the data is put into data
// ...
dvector boot_sample=bootstrap(data,200,1231); // Make a sample of size 200
                          // and create the dvector boot_sample to hold it
// ...
```

While sorting is not strictly a part of methods for calculating the derivatives of differentiable functions (it is a highly non-differentiable operation) it is so useful for pre- and post-processing data that we have included some functions for sorting `dvector` and `dmatrix` objects. If `v` is a `dvector` the statement

```
dvector w=sort(v);
```

will sort the elements of `v` in ascending order and put them in the `dvector` object `w`. The minimum and maximum valid indices of `w` will be the same as those of `v`. If desired an index table for the sort can be constructed by passing and `ivector` along with the `dvector`. This index tables can be used to sort other vectors in the same order as the original vector by using the `()` operator.

```
dvector u={4,2,1};
dvector v={1,6,5}
ivector ind(1,3);
dvector w=sort(u,ind); // ind will contain an index table for the sort
// Now w=(1,2,4) and ind=(3,2,1)
dvector ww=v(ind);     // This is the use of the ( ) operator for subset
                       // selection.
// Now ww=(5,6,1)
```

The sort function for a `dmatrix` object sort the columns of the dmatrix into ascending order, using the column specified to do the sorting. For example

```
dmatrix MM = sort(M,3);
```

will put the sorted matrix into `MM` and the third column of `MM` will be sorted in ascending order.

The following functions have been included in AUTODIF by overloading the C++ library functions

```
sin cos tan asin atan acos sinh cosh tanh fabs (sfabs) exp log log10 sqrt pow
```

These functions can be used on numbers or vector_objects in the form

```
number = function(number);
vector_object = function(vector_object);
```

When operating on `vector_objects` the functions operate element by element, so that if `y` is a `dvector` whose elements are $(y_1, \ldots, y_n)$ then `exp(y)` is a `dvector` whose elements are $(\exp(y_1), \ldots, \exp(y_n))$.

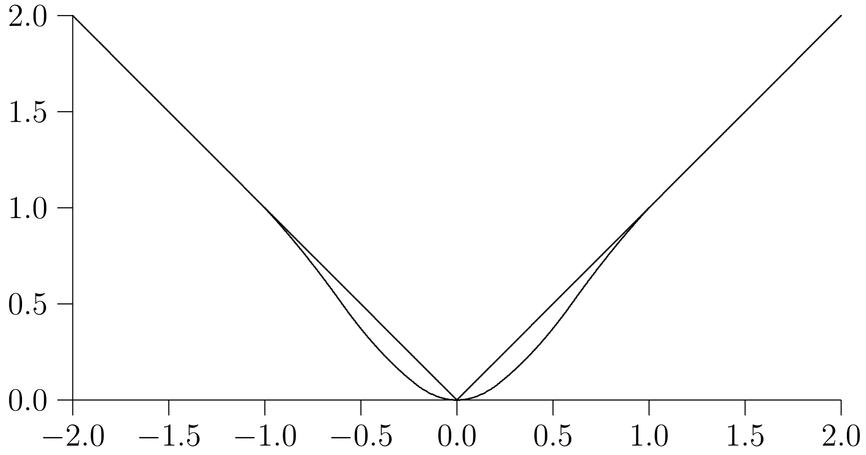
The functions `min` and `max` when applied to a `vector_object` return a `number` which is equal to the minimum or maximum element of the `vector_object`

The function `sfabs` is a "smoothed" absolute value function which agrees with the standard absolution function for all values of its argument with absolute value greater than 0.001. For values of its argument less than 0.001 in absolute value the function uses cubic splines to produce a differentiable approximation to the absolute value function. This function is intended to be used when a differentiable function which approximates the absolute value function is needed.

While the function `sfabs` has been set to agree with the absolute value function `fabs` for all values greater than 0.001 value has it is simple matter to rescale the function to that it will agree with the absolute value function over any desired range by multiplying and dividing it by the desired scale factor.

```
double scale
double x
// ...
double v=scale*sfabs(x/scale); // This function will agree with fabs
                               // for all components of w with absolute value
                               // greater than scale*.001
```

Figure 4.1 Plot of the smoothed absolute value function `y=1000*sfabs(x/1000)` and the normal absolute value function `y=fabs(x)`



In figure 4.1 the function `sfabs` has been "magnified" by a scale factor of 1000 to show how it differs from the ordinary absolute value function `fabs`.

Chapter 5

Advanced concepts

This chapter is devoted to various matters which enable the AUTODIF user to create more advanced applications and to obtain faster performance from the AUTODIF code created. Begining users can skip or briefly skim the topics contained here.

A disadvantage of the reverse mode of automatic differentiation is the large amount of temporary storage required for keeping the intermediate data required for the derivative calculations. Every arithmetic operation on a `dvariable` object generates 28-30 bytes of temporary storage. To see what this means consider that to invert an $n \times n$ matrix requires about n^3 such operations, so that the temporary storage required for inverting a 100 by 100 matrix would be about 28-30 megabytes.

To reduce the requirements for temporary storage AUTODIF employs precompiled derivative code for common operation on variable container classes. Depending on the operation this can greatly reduce the temporary storage used. For the matrix inverse the storage for a 100 by 100 matrix is about 500K, a reduction of about 60 to 1. For a vector dot product the reduction in temporary storage requirements is about 3.5 to 1. While this is much less than the matrix inverse it can still be a considerable saving in large problems.

To achieve the best performance you should use AUTODIF's vector and array operations on variable objects whenever possible rather than writing your own code to manipulate elements of these container classes. Consider the following code fragment which "normalizes" a `dvar_vector` so that its components sum to `1`.

```
dvar_vector v(1,n);
// get some data into v somehow
// ...
// now normalize v so that its components sum to 1
dvariable sum=0.;
for (int i=1;i<=n;i++)
{
  sum+=v(i);
}
```

```
for (i=1;i<=n;i++)
{
  v(i)=v(i)/sum;
}
```

The same normalization can be achieved by the following code which uses the AUTODIF function `sum` which sums the components of a vector object as well as the divide operator for dividing a vector by a number. It will produce less temporary derivative information.

```
dvar_vector v(1,n);
// get some data into v somehow
// ...
// now nomalize v so that its components sum to 1
v=v/sum(v);
```

Even more efficient code is generated by employing the `/=` operator which will divide a vector by a number.

```
v/=sum(v);
```

Consider the following code fragment for adding two matrix objects

```
dvar_matrix a(1,n,1,n);
dvar_matrix b(1,n,1,n);
dvar_matrix c(1,n,1,n);
// ...
for (int i=1;i<=n;i++)
{
  for (int j=1;j<=n;j++)
  {
    a(i,j)=b(i,j)+c(i,j);
  }
}
```

A FORTRAN compiler which only has to deal with a small number of array types can be expected to produce well optimized code from the above source code. C++ compilers are less likely to be able to exploit that fact that the first index `i` is constant in the inner loop. It is to be expected that in the future the ability of C++ compilers to optimize code will improve. In the meantime code segments like the above can be optimized by the user through the use of "references". If `a` is a `dvar_matrix` then `a(i)` (or `a[i]`) is a `dvar_vector`. This fact can be used to reduce the redundant adressing operations on the first index in the above example by defining reference objects for these `dvar_vectors`. If you are not familiar with the concept of references which are extremely useful in C++, consult your C++ reference manual.

```
dvar_matrix a(1,n,1,n);
dvar_matrix b(1,n,1,n);
dvar_matrix c(1,n,1,n);
// ...
for (int i=1;i<=n;i++)
{
  dvar_vector& ai=a(i);
  dvar_vector& bi=b(i);
  dvar_vector& ci=c(i);
  for (int j=1;j<=n;j++)
  {
    ai(j)=bi(j)+ci(j);
  }
}
```

Note however that the most efficient way to add two matrices is to use the code

```
a=b+c;
```

Of course there is some overhead involved in establishing the references so that this would not be worthwhile for very small values of `n`.

A big advantage of C++ over traditional scientific programming languages such as FORTRAN is the increased facility for defining and manipulating complex data structures. Many real world situations can not be conveniently described in terms of the simple "flat" data structures represented by vectors, matrices, and regular three dimensional arrays.

As an aid to creating more complex data structures AUTODIF's two and three dimensional container classes have been extended to what we call ragged arrays. In short, a ragged matrix is a matrix whose rows consist of vectors of different lengths and varying valid index ranges, while a ragged three dimensional array can be viewed as an array of matrices of different sizes while the matrices which make up the array may or may not be ragged themselves.

To specify a ragged matrix requires two integers for the minimum and maximum valid row indices, a vector of integers for the minimum valid index of the vector forming each row of the matrix, and a vector of integers for the maximum valid index of the vector forming each row of the matrix.

```
int min=0;
int max=4;
ivector minind(0,4);
ivector maxind(0,4);
// get the desired values into the ivectors minind and maxind
// ...
dmatrix M(min,max,minind,maxind);
```

Notice that the minimum and maximum valid indices for `minind` and `maxind` must be the same as the minimum and maximum valid row indices for the matrix `M`.

Suppose that `minind(3)=-1` and `maxind(3)=5`. Then the fourth row of `M` will be a vector with seven components whose minimum valid index is `-1` and whose maximum valid index is `5`.

Often one desires to make a ragged matrix where the minimum valid index for all the rows is the same. This can be accomplished by filling the ivector `minind` with the same number in each component. For convenience a matrix constructor which takes an `int` for its third argument has been supplied.

```
int min=0;
int max=4;
int minind=1;
ivector maxind(0,4);
// get the desired values into the ivector maxind
dmatrix M(min,max,minind,maxind);
```

This will make a ragged matrix `M`. The minimum valid indices for vectors making up the rows of `M` will all be equal to `1`.

The range of constructors available for ragged three dimensional arrays is considerably more complicated than for ragged matrices. The most general constructor for a ragged three dimensional array has the following prototype.

```
d3_array(int sl,int su,ivector& rl,ivector& ru,imatrix& cl,imatrix& cu);
```

The i'th slice of the `d3_array` is a ragged `dmatrix` which is constructed using the data `rl(i) ru(i) cl(i) cu(i)` as described above. Notice that `rl(i) ru(i)` are of type `int` while `cl(i) cu(i)` are objects of type `ivector` so that this all makes sense. This is the most "ragged" of three dimensional objects. There is also a constructor with the following prototype

```
d3_array(int sl,int su,ivector& rl,ivector& ru,ivector& cl,ivector& cu);
```

The i'th slice of the `d3_array` will use the data `rl(i) ru(i) cl(i) cu(i)` to make a `dmatrix`. Since these arguments are all of type `int` each slice of the ragged three dimensional array consists of a "non-ragged" or regular `matrix` object. An example of the use of a ragged three dimensional array for a data structure is given in the chapter on neural networks. The complete list of constructors provided for specifying three dimensional arrays is:

```
d3_array( int& sl,  int& sh,  int& nrl,
   int& nrh, int& ncl,  int& nch);

d3_array(int& sl,int& sh,ivector& nrl,ivector& nrh,
  imatrix& ncl,imatrix& nch);

d3_array(int& sl,int& sh,ivector& nrl,ivector& nrh,
  int ncl,imatrix& nch);

d3_array(int& sl,int& sh,ivector& nrl,ivector& nrh,
  ivector& ncl,ivector& nch);

d3_array(int& sl,int& sh,int& nrl,ivector& nrh,
  int& ncl,ivector& nch);
```

```
    d3_array(int& sl,int& sh,int& nrl,ivector& nrh,
      int& ncl,int& nch);

    d3_array(int sl,int sh,int nrl,ivector& nrh,
      int ncl,imatrix& nch);
```

As your model becomes more complex the number of AUTODIF objects required will increase until the code again starts to become unmanageable. One solution is to group these objects themselves into structures or classes so that they can be manipulated as single objects.

Suppose that your model has 5 vector objects whose minimum valid indices are 1 and whose maximum valid indices are all equal to an integer n whose value is known at run time. Denote these vector objects by a1,a2,...,a5. Let the model have 5 more vector objects whose minimum valid indices are 1 and whose maximum valid indices are 2n. Denote these vector objects by b1,b2,...,b5. Finally suppose the model has a (regular) matrix object whose valid row indices go from 1 to n and whose valid column indices go from 0 to 2n. Denote this matrix by M. We can encapsulate all these objects in a class as follows.

```
// class definition for constant_model_parameters
class constant_model_parameters
{
public:
  int size;
  dvector a1;
  dvector a2;
  dvector a3;
  dvector a4;
  dvector a5;
  dvector b1;
  dvector b2;
  dvector b3;
  dvector b4;
  dvector b5;
  dmatrix M;
  constant_model_parameters(int n); // prototype for constructor
};
```

At the point in your program where you want to “create” this object you will need to invoke the constructor described below. The invocation is simply a declaration of the object with its name such as:

```
int n=10;
// ...
constant_model_parameters cmp(n); //creates the object with name "cmp"
```

The main part of the implementation of the class is in the definition of the constructor.

```
 constant_model_parameters::constant_model_parameters(int n) :
   a1(1,n),a2(1,n),a3(1,n),a4(1,n),a5(1,n),
   b1(1,2*n),b2(1,2*n),b3(1,2*n),b4(1,2*n),b5(1,2*n),
   M(1,n,0,2*n) {size=n;}
```

If you are not familiar with the form of the constructor above you should consult a C++ reference book. The idea is that the constructors for all the classes which are members of the class constant_model_parameters are invoked after the : and before the body of the constructor.

Note that while for the "safe" library all container classes are initialized to 0, for the optimized library they are not. If you want to initialize some of these objects, such as a1 and M this should be done in the body of the constructor.

```
constant_model_parameters::constant_model_parameters(int n) :
  a1(1,n),a2(1,n),a3(1,n),a4(1,n),a5(1,n),
  b1(1,2*n),b2(1,2*n),b3(1,2*n),b4(1,2*n),b5(1,2*n),
  M(1,n,0,2*n)
{
  size=n;
  a1.initialize();  // set components of a1 and M to 0
  M.initialize();
}
```

All the objects in the class constant_model_parameters are as the name implies, constants. This class could be useful for reading in the parameters from a file, for example, and otherwise manipulating them. At some point presumably the user wants the parameters transformed into variable objects so that derivative information can be generated. This can be done by defining a corresponding variable class.

```
// class definition for variable_model_parameters
class variable_model_parameters
{
public:
  dvar_vector a1;
  dvar_vector a2;
  dvar_vector a3;
  dvar_vector a4;
  dvar_vector a5;
  dvar_vector b1;
  dvar_vector b2;
  dvar_vector b3;
  dvar_vector b4;
  dvar_vector b5;
  dvar_matrix M;
  // This constructor transforms the constant parameters into
  // variable parameters
  variable_model_parameters(constant_model_parameters&); // prototype
                                                         // for constructor
};
```

The constructor for the class variable_model_parameters takes the constant objects in the class constant_model_parameters and converts them to variable objects.

```
variable_model_parameters::variable_model_parameters
  (constant_model_parameters& cms) :
  a1(cms.a1),a2(cms.a2),a3(cms.a3),a4(cms.a4),a5(cms.a5),
  b1(cms.b1),b2(cms.b2),b3(cms.b3),b4(cms.b4),b5(cms.b5),
  M(cms.M) {size=cms.size;}
```

After all this has been done the constant and variable classes could be used something like

```
void fcomp(variable_model_parameters&); // function prototype for fcomp
int n=10;
constant_model_parameters cms(n);
// get some data into cms
//
// call a function which takes an object of type variable_model_parameters
fcomp(cms); // cms will be converted into an object of type
            // variable_model_parameters
```

Since in this example, the only constructor which has been defined for an object of type variable_model_parameters takes an argument of type constant_model_parameters the only way to create the variable object will be to first create the constant object. This may or may not be what is required. Of course a constructor can be defined which will create a variable object from the int n.

You should be aware that the C++ compiler attempts to create default constructors for complex classes if they are not provided by the user. To do this it employs the constructors provided for the members of the complex class. For examples if you employ the lines of code

```
constant_model_parameters cmp(10);
constant_model_parameters cmp1=cmp;
```

the compiler will construct a default copy constructor for the class constant_model_parameters (assuming that you, the user, have not already defined one). In order to create the default copy constructor the compiler will invoke the copy constructors provided by AUTODIF for dvector and dmatrix classes. Since these constrtuctors perform shallow copies, the dvector and dmatrix objects in cmp1 will not be distinct from the corresponding objects cmp. If you want to define a copy constructor for the class constant_model_parameters which will create a distinct object it could be done as follows:

```
constant_model_parameters::
  constant_model_parameters(constant_model_parameters cms) :
  a1(1,cms.size),a2(1,cms.size),a3(1,cms.size),a4(1,cms.size),
  a5(1,cms.size),b1(1,2*cms.n),b2(1,2*cms.n),b3(1,2*cms.n),
  b4(1,2*cms.n),b5(1,2*cms.n),
  M(1,cms.size,0,2*cms.size)
{
  size=cms.size;
  a1=cms.a1;
  a2=cms.a2;
  a3=cms.a3;
  a4=cms.a4;
  a5=cms.a5;
  b1=cms.b1;
  b2=cms.b2;
  b3=cms.b3;
  b4=cms.b4;
  b5=cms.b5;
  M=cms.M;
}
```

Chapter 6

The AUTODIF libraries

AUTODIF is supplied with safe and optimized versions of its object module libraries. The safe version is intended to be used for code development. It provides for bounds checking of arrays and initialization of all declared objects. This initialization sets the value of all array members to 0 and initializes the derivative structure associated with each variable type object. The initialization of the derivative structures can involve a fair amount of overhead so that after a project has been debugged the user may wish to use the optimized version of the library. The optimized version does not provide array bounds checking or initialization of objects. It is recommended that the user initializes all objects which require initialization during the development stage of the program even though this is not necessary when using the safe library so that the project code can be easily linked with the optimized library later if desired. If an application runs successfully using the safe library, but does not run properly using the optimized library, there is probably an unitialized object being used somewhere in the code.

The names and number of libaries supplied with AUTODIF vary from compiler to compiler, so check the `read.me` file on the distribution disk for the names of the libraries supplied with your version of AUTODIF.

The class member functions `initialize` are provided for initializing all appropriate AUTODIF classes. For example to declare and initialize a dvar_matrix M you could use the following code fragment.

```
dvar_matrix M(1,20,1,30) // Creates a 20 by 30 matrix
M.initialize();          // Sets all the elements of M to 0 and initializes
                         // the derivative structures for M
```

An example of code where an object requires initialization is

```
{
  // . . .
  dvariable u; //u is declared but not initialized
  for (int i=1;i<=n;i++)
  {
    u+=x[i];  // x is a vector object which was previously declared
  }            // This is an error for the optimized library
 }
```

In this example the `dvariable u` is declared, but not initialized. The use of the unitialized `u` in the expression `u+=x[i]` is an error if the code is linked with the optimized library. It will perform fine if the code is linked with the safe library. To use the optimized library this code could be rewritten as

```
{
  // . . .
  dvariable u=0; //u is declared and initialized
  for (int i=1;i<=n;i++)
  {
    u+=x[i];  // x is a vector object which was previously declared
  }
}
```

The code could also be written as

```
{
  // . . .
  dvariable u; //u is declared but not initialized
  u=0;         //u is initialized
  for (int i=1;i<=n;i++)
  {
    u+=x[i];  // x is a vector object which was previously declared
  }
}
```

It is slightly less efficient to initialize x in this way.

For the BORLANDC++ compiler both libraries have been compiled to enable overlays.

Chapter 7

Input and Output

Input and output methods for the AUTODIF class objects have been implemented by overloading of the C++ stream I/O operators `<<` and `>>` for all AUTODIF classes. The so-called *iostream* package, introduced in AT& T release 2.0 of C++ is fully supported. For a detailed discussion of C++ stream I/O you should consult your C++ manual (and the appropriate header files). In this section, we discuss extension of stream I/O to AUTODIF class objects. Formatted input and output uses the standard C++ stream classes, `istream`, `ostream`, `ifstream`, and `ofstream`. In addition, unformatted ("binary") file I/O is implemented through two additional stream classes `uostream` and `uistream`.

Variables of type `dvariable, dvar_array, dvar_matrix, dvector,` and `dmatrix` may be input or output using the standard stream operators and formatted using the standard manipulators. A two additional manipulators are provided to simplify selection of either fixed point or "scientific" (e-format) notation. The use of the manipulators `setw()` (set width), `setprecision()`, `setfixed`, `setscientific` and `endl` is illustrated below

```
#include <iomanip.h>
#include <fvar.hpp>
main()
{
  int n = 30;
  int m = 5;
  gradient_structure gs;
  dvar_matrix x(1, n, 1, m);
  .
  .   // code to calculate x
  .
  // e format with default width and precision
  cout << setscientific << x << endl;

  // fixed point format
  cout << setw(13) << setprecision(3) << setfixed << x << endl;
  .
  .
  .
}
```

The first output statement will produce thirty lines with five numbers per line in "scientific" (e-format) separated by one or more blanks. The second output statement will produce a table with 30 rows and 5 columns with each member of `x` printed with a fixed decimal place with 3 significant figures in a field 13 bytes wide. If the manipulators are omitted, the output will appear with the default width, precision, and format specified by the compiler and with each number separated by one spaces.

The following standard manipulators are also supported:

```
setf(f, ios::floatfield);  // to select fixed or scientific
setf(b, ios::basefield);   // to base of number
setf(a, ios::adjustfield); // to select justification of fields
// where
//   f  may be ios::fixed or ios::scientific
//   b  may be ios::dec, ios::oct or ios::hex
//   a  may be ios::left, ios::right or ios::internal
```

Although the stream operators `>>` and `<<` make it very easy to perform input or output on the AUTODIF container classes, it is important to remember that they do not perform any error checking. While a complete discussion of error states for the stream classes is beyond the scope of this manual a brief discussion of the use of the operator `!` for I/O error checking is included. If `infile` is a stream object then after any I/O operation involving `infile`, `!infile` will be true if the operation has failed and false if the operation has succeeded. For example:

```
dvector x(1,10);
ifstream infile("data"); // associate the file data with the ifstream
                         // object infile
if (!infile) // Check if the file has been successfully opened
{
  cerr << "Error trying to open file data\n";
  exit(1); // User has decided to stop on this error condition
}

infile >> x ;            // read data into the dvector x;
if (!infile) // Check if the input operation was successful
{
  cerr << "Error trying to read dvector x from file data\n";
  exit(1); // User has decided to stop on this error condition
}
```

If you neglect to put in any error checking for I/O operations, your program may well carry on blissfully even though the I/O operation has failed. The symptoms of this failure can be quite confusing and waste a lot of valuable program development time.

It is often useful to store intermediate calculations in an external file. Unformatted ("binary") file I/O is the most rapid and space efficient method for such operations and is supported through two additional stream classes `uostream` and `uistream`. These two classes are derived classes from the `ofstream` and `ifstream` classes in the *iostream* package.

The following listing illustrates the main features of the AUTODIF system I/O support. It also illustrates the use fill operations and C++ scoping rules to automatically destroy some variables. `#include <fvar.hpp>`

```
// compiler specific stack length specifications
#ifdef __BCPLUSPLUS__
  #include <iomanip.h>
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  #include <iomanip.hpp>
  long _stack = 20000;
#endif
void main()
{
  int n = 20;
  int m = 5;
  int i, j;
  {
    gradient_structure gs;
    dvar_matrix x(1, n, 1, m);
    // compute some values for x
    for (i=1; i<=n; i++)
    {
      x.rowfill_seqadd(i,i+1./1000.,1./1000.);
    }

    // display x on screen
    cout << "Prior to write and read tests\n";
    cout << "  In default format:\n";
    cout << x << endl;
    cout << "  In fixed point notation:\n";
    cout << setw(13) << setprecision(4) << setfixed << x << endl;
    cout << "  In scientific notation:\n";
    cout << setw(13) << setprecision(4) << setscientific << x << endl;

    // write x to formatted file, read back into y and display on screen
    cout << "After formatted write and read:\n";
    {
      ofstream f1("file1.tmp");  // open output file
      if (!f1)
      {
        cerr << "Error trying to open file file1.tmp\n";
        exit(1);
      }

      f1 << x;   // write x to file with default format
      if (!f1)
      {
        cerr << "Error trying to write x to file1.tmp\n";
        exit(1);
      }
    }  // f1 goes out of scope so file1.tmp is closed
    {
      dvar_matrix y(1, n, 1, m);
      ifstream f1( "file1.tmp");  // open input file
      if (!f1)
      {
        cerr << "Error trying to open input file file1.tmp\n";
        exit(1);
      }
      f1 >> y;  // read back into y
      if (!f1)
      {
        cerr << "Error trying to read y from file1.tmp\n";
        exit(1);
      }
      cout << setw(14) << setprecision(4) << y << endl;
    }  // y and f1 go out of scope; file1.tmp is closed

    // write x to unformatted file, read back into y and display on screen
    cout << "After unformatted write and read:\n";
    {
      uostream f1("file2.tmp");  // open output file
      if (!f1)
      {
        cerr << "Error trying to open unformatted output file file1.tmp\n";
        exit(1);
      }
      f1 << x; // write x to file
      if (!f1)
      {
        cerr << "Error trying unformatted write x to file1.tmp\n";
        exit(1);
      }
    } // f1 goes out of scope so file2.tmp is closed
    {
      dvar_matrix y(1, n, 1, m);
      uistream f1( "file2.tmp");  // open input file
      f1 >> y;  // read back into y
      cout << setw(14) << setprecision(4) << y << endl;
    } // y and f1 go out of scope; file2.tmp is closed
  } // x and gs go out of scope
  exit(0);
}
```

Chapter 8

Temporary Files

The AUTODIF system saves information required to calculate the derivatives on a stack during the evaluation of the function. The amount of information depends on the number of variables for which derivative information is required and on the specific nature of the computations. It is difficult to predict generally how much information will be generated. If the length of this stack exceeds a preset amount, it is stored on the disk in one or two unformatted temporary gradient files, `gradfil1.tmp` and `gradfil2.tmp`. There is also derivative information from "precompiled" derivative calculations which is stored in the file `cmpdiff.tmp`.

The location of these files is controlled by the DOS environment strings, `TMP` and `TMP1`. The file `gradfil1.tmp` will be created in the directory indicated by `TMP` while the file `cmpdiff.tmp` will be created in the directory indicated by `TMP1`. If either or both of these strings is undefined the current directory will be used to create the files. Greatest speed will be achieved if the temporary files are located on a RAM disk. Assuming that your RAM disk is drive `E:`, type `set tmp=e:` at the DOS prompt. If `gradfil1.tmp`, becomes too large for your RAM disk (or whatever device is specified by the `TMP` environment string), the rest of the file will be stored in `gradfil2.tmp` on the device referred to by the DOS environment string `TMP1`. For example if you have typed `set tmp=e:` and `set tmp1=c:\temp`, the first part of the gradient information will be stored on `e:` (presumably a RAM disk) while the second part will be stored in the directory `c:\temp`. If the directory does not exist, AUTODIF will display an error message and retire. If either the `TMP` or `TMP1` environment string does not exist AUTODIF will create `gradfil1.tmp` or `gradfil2.tmp` in the current directory.

The temporary gradient files are normally deleted when objects of class `gradient_structure` go out of scope. Abnormal termination of the program (such as by pressing `Ctrl-Break` or use of the derivative checker) will cause some temporary gradient files to remain in your file system. These files may become quite large, several Mb or more, so it is a good idea to remove them.

Chapter 9

Global Variables

The AUTODIF system declares a number of global variables which control memory allocation for the AUTODIF structures. These global variables can be changed by the use of "access functions" prior to declaring an object of type gradient_structure. For most applications the default values of these global variables should suffice. You will not need to worry about them until you start building larger applications.

```
int NUM_RETURN_ARRAYS = 10;
```

This global variable determines the maximum allowable depth of nesting of functions which return AUTODIF variable types (see the discussion about the RETURN_ARRAYS_INCREMENT() and RETURN_ARRAYS_DECREMENT() instructions). The default value of 10 will suffice for almost all applications and most users will not need to concern themselves with this variable. To increase the number to 15 you should put the instruction

```
gradient_structure::set_NUM_RETURN_ARRAYS(15);
```

into your program before an object of type gradient_structure is declared to manage the derivative calculations. If you exceed the maximum allowabel depth of nesting the message

```
Overflow in RETURN_ARRAYS stack -- Increase NUM_RETURN_ARRAYS
```

will appear. This message can also occur if you have put a RETURN_ARAYS_INCREMENT which is not matched by a RETURN_ARAYS_DECREMENT into one of your functions.

```
long int GRADSTACK_BUFFER_SIZE = 2200;
```

This global variable determines the number of entries which are contained in the buffer which contains the information necessary for caclulating derivatives. For historical reasons the actual amount of memory reserved for the buffer in bytes is equal to the value of GRADSTACK_BUFFER_SIZE multiplied by the size in bytes of an AUTODIF structure, grad_stack_entry. For 16 bit DOS compilers the size of grad_stack_entry is 28 bytes. The default value of GRADSTACK_BUFFER_SIZE is 2200 which reserves a buffer of 28*2200=61600 bytes. Since for 16 bit DOS versions of AUTODIF buffers are limited in size to a maximum of 64K, 2200 is about the largest size which can be used.

If you make the value smaller (say 500) it will free up some memory (about 50K), but the program will execute more slowly because it must store data on the hard disk more often. This is a desperation move in a situation where you must find some more memory for your program. To decrease the number to 500 you should put the instruction

```
gradient_structure::set_GRAD_STACK_BUFFER_SIZE(500);
```

into your program before declaring a `gradient_structure` object.

```
long int CMPDIF_BUFFER_SIZE = 32000L;
```

This global variable determines the size in bytes of the buffer used to contain the information generated by the “precompiled” derivative code. To change this variable the instruction

```
static void gradient_structure::set_CMPDIF_BUFFER_SIZE(long int i);
```

is used.

```
int RETURN_ARRAYS_SIZE = 50;
```

This global variable controls the amount of complexity which one line of arithmetic can have. The present default value should be large enough for any conceivable purpose. It is recommended that the user leave it alone.

```
int MAX_NVAR_OFFSET = 200;
```

This global variable determines the maximum number of independent variables which can be used. In can be increased, for example, to 500 by including the instruction

```
gradient_structure::set_MAX_NVAR_OFFSET(500);
```

into your program before declaring a `gradient_structure` object. If you have more independent variables than the value of `MAX_NVAR_OFFSET` the error message

```
You need to increase the global variable MAX_NVAR_OFFSET to xxx
```

will appear and execution will be halted.

```
unsigned long ARRAY_MEMBLOCK_SIZE=100000L;
```

This global variable determines the maximum amount of memory (in bytes) available for AUTODIF variable type container class objects (`dvar_vector, dvar_matrix ... etc`). It represents the largest single amount of memory which is required when an object of type `gradient_structure` is declared. An AUTODIF variable array object requires about 8 bytes for each element of the array. A `dvar_vector` object of size 500 will require slightly more than 4000 bytes of memory. A `dvar_matrix` object with 20 rows and 30 columns would require 20 times 30 times 8 = 4800 bytes of memory. To provide the maximum amount of memory for other purposes you can reduce the size of ARRAY_MEMBLOCK_SIZE until an error message telling you to increase ARRAY_MEMBLOCK_SIZE appears.

```
unsigned MAX_DLINKS  = 1000;
```

The access function

```
gradient_structure::set_MAX_DLINKS(int i);
```

determines the maximum number of dvariable objects which can be in scope at one time. The default value is 1000 which should be enough for most purposes. If the value is exceeded the message

```
Need to increase the maximum number of dlinks
```

will appear and program execution will be terminated.

The complete list of access functions for the AUTODIF system global variables is:

```
static void set_NUM_RETURN_ARRAYS(int i);
static void set_ARRAY_MEMBLOCK_SIZE(unsigned long i);
static void set_GRADSTACK_BUFFER_SIZE(long int i);
static void set_MAX_NVAR_OFFSET(unsigned int i);
static void set_MAX_DLINKS(int i);
static void set_CMPDIF_BUFFER_SIZE(long int i);
```

These functions may not be used while an object of type gradient_structure is in scope. Doing so will produce an error message and program execution will be halted.

The size of the stack is usually controlled by a compiler-specific global variable. The default stack size is generally too small for AUTODIF. It is recommended that the stack size be increased to 15000 or 20000 bytes. The AUTODIF library routines have been compiled without the option of checking for stack overflow so that if you do not reserve enough area for the stack, your program will probably crash the system without any warning or simply behave in some strange manner.

Stack size is set by inserting an appropriate line in the code prior to the main() block. The following examples set the stack length to 15000 bytes. Refer to your compiler documentation for more details.

For the Borland compiler, the appropriate syntax is:

```
extern unsigned _stklen = 15000;
main()
{
  // ... your code
}
```

For the Zortech compiler, the appropriate syntax is:

```
unsigned int _stack = 15000;
main()
{
  // ... your code
}
```

Chapter 10

The AUTODIF function minimizing routines

AUTODIF is supplied with both a quasi-Newton function minimizer and a conjugate gradient function minimizer. A quasi-Newton function minimizer is an effective method for minimizing smooth functions of up to one or two hundred parameters. At some point, the calculations involved in updating the inverse Hessian approximation (on the order n^3) and the memory requirements for the inverse Hessian approximation ($n(n+1)/2$ floating point elements), where n is the number of parameters, become prohibitive and the conjugate gradient method, supplied with AUTODIF, should be used.

We have provided the user with two types of interfaces to the function minimizer routines. The first approach provides the user with the maximum amount of encapsulation of the minimization routine. It effectively allows the user to say "Here is my function. Minimize it." The disadvantage of this approach is some loss of flexibility in managing memory and controlling the minimization process. To provide maximum flexibility we also provide a lower level interface which allows the user to exercise complete control over the minimization process.

The fully encapsulated minimization procedure is invoked by calling the function `minimize`. The following code invokes the `minimize` routine to find the minimum for the user's function `userfun`. `//file: minimize.cpp`

```
#include <fvar.hpp>

double userfun(dvar_vector&);

#ifdef __BCPLUSPLUS__
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  long _stack = 20000;
#endif
void main()
{
  int nvar=60;
```

```
    independent_variables x(1,nvar);
    fmm fmc(nvar); // creates the function minimizing control structure
    double minimum_value= fmc.minimize(x,userfun); //Call the function minimizer
    cout << "The minimum value = " << minimum_value << "at x =\n"
      << x << "\n";
  }
```

In this example, it is the user's responsibility to determine the number of independent variables nvar and to declare the array of type independent_variables which holds the independent variables. The elements of the independent_variables array x are initialized to 0. If other initial values are desired, the user must initialize the array as well. An example of a typical simple function which could be minimized by this code is the code for the function

$$(x[1]-1)^2+\sum_{i=1}^{i<n}(x[i+1]-x[i])^2$$

whose minimum is at the point $(1,1,\ldots,1)$.

```
//file: userfin.cpp
#include <fvar.hpp>
double userfun(dvar_vector& x)
{
  dvariable z,tmp;
  int min,max;
  min=x.indexmin();
  max=x.indexmax();
  tmp=x[min]-1;
  z+=tmp*tmp;
  for (int i=min;i<max;i++)
  {
    tmp=x[i+1]-x[i];
    z+=tmp*tmp;
  }
  return(value(z));
}
```

While the vector of independent variables is declared to be an object of type independent_variables in the main routine it must be declared to be an object of type dvar_vector in userfun. This is the AUTODIF "array" type for which derivative information is calculated. It corresponds to an array or vector of doubles. Also, the quantities z and tmp which correspond to objects of type double for which derivative information is required must be declared to be of type dvariable. The function value takes a dvariable as its argument and returns a double which is the value of the dvariable.

This simple implementation of the function minimizer is useful for the beginning user who doesn't wish to get too involved in the intricacies of AUTODIF or who wants a quick solution to a minimization problem. What are its disadvantages? The main disadvantage is that the type of arguments taken by the function userfun are limited to a predefined set. In the present case the function userfun is declared to be a function which takes one argument of type dvar_vector&. The line of code

```
double minimum_value= minimize(x,userfun);
```

causes a pointer to the function `userfun` to be passed to the function `minimize` (when a function is named without the accompanying parentheses a pointer to the function is generated). Thus the second argument of `minimize` is a pointer to a function which takes one argument of type `dvar_vector&` and returns a `double`.

To enable the user to retain maximum control over memory management and the minimization process, the function minimizer and the function being minimized should be put on the same level in the program. This is accomplished by putting them into a loop. The general form of this control structure is

```
double f;
independent_variables x(1,nvar);
dvector g(1,nvar);
fmm fmc(nvar);  // Use this declaration for the quasi-Newton function minimizer
// fmmc fmc(nvar);  // Use this declaration for the conjugate-gradient function
                    //minimizer
gradient_structure gs;
while (fmc.ireturn >=0)
{
  fmc.fmin(f,x,g);
  if (fmc.ireturn > 0)
  {
    f=userfun(x,...) // The users function
    gradcalc(nvar,g);
  }
}
```

The `dvector` objects `g` and `x` must have minimum valid index equal to 1 and maximum valid index equal to the number of independent variables in the minimization.

Since many parts of the code sequence are independent of the particular application the code segments for calling the function minimizer and evaluating the derivatives has been made into a macro. Using the macros, `BEGIN_MINIMIZATION(nvar,f,x,g,fmc)` and `END_MINIMIZATION(nvar,g)` the above code can be written as

```
double f;
independent_variables x(1,nvar);
dvector g(1,nvar);
fmm fmc(nvar); // Use this declaration for the quasi-Newton function minimizer
// fmmc fmc(nvar);  // Use this declaration for the conjugate-gradient function
                    //minimizer
BEGIN_MINIMIZATION(nvar,f,x,g,fmc)
  f=userfun(x,...) // The users function
END_MINIMIZATION(nvar,g)
```

The macro `BEGIN_MINIMIZATION` takes five arguments, the number of independent variables, the name of the object of type `double` which will contain the value of the function being minimized, the independent variables, the gradient vector, and the name of the function minimizing control structure. The macro, `END_MINIMIZATION` takes two arguments, the number of independent variables and the gradient vector.

Notice that even when using the macros, it is the responsibility of the user to declare the name of the object of type `double`, `f`, which will contain the value of the function being minimized, the independent_variables x, and the dvector g as well as the fmm object fmc. The quasi-Newton function minimizer fmin is controlled by and is a member function of the class fmm. The constructor for the class fmm takes nvar, the number of variables, as an argument and creates the structure, fmc which controls the minimization. The user can control the operation of the function minimizer by changing certain values of the fmm structure fmc. The conjugate gradient function minimizer fmin is a member function of the class fmmc. The only difference in calling the quasi-Newton or conjugate-gradient function minimizers is in the declaration of an object of type fmm for the quasi-Newton method and of type fmmc for the conjugate-gradient method. The public members of this structure are given below.

```
class fmm
{
public:
                        // control variables:

  long   maxfn;        // maximum number of function evaluations
                       // default value: maxfn = 500
  double min_improve;  // if the function doesn't decrease by at least this
                       // amount over 10 iterations then quit.
                       // default value: min_improve=1.e-6
  double crit;         // gradient convergence criterion; fmin terminates if
                       //   maximum absolute value of the
                       //   gradient component is less than crit.
                       //   default value: crit = 1e-4
  long   imax;         // maximum number of function evaluations in linear
                       // search without improving the current estimate
                       // before fmin terminates  default value : imax = 30
  long   iprint;       // number of iterations between screen displays of
                       // current parameter estimates default value: iprint = 1
  long   scroll_flag;  // set to 1 to scroll display on screen;
                       //        0 to overwrite screen
                       // default value: scroll_flag = 1
  int    ireturn;      // used to control the loop containing fmin and the
                       // user's function
}
```

There are five criteria by which the function minimizer may decide that convergence has been achieved.

1. If the magnitude of all the gradient components is less than crit (default value 1.e-4).

2. If the maximum number of function evaluations, maxfn is exceeded (default value 500)

3. If the function value does not decrease by more than min_improve over 10 iterations (default value 1.e-6).

4. If the function minimizer tries to evaluate the function more than imax times within one linear search. (default value 30)

5. If the function minimizer detects that the user has pressed the Q or q key.

Here is an example of the use of these switches.

```
dvector g(1,nvar);
dvector x(1,nvar);
double f;
fmm fmc(nvar);
fmc.maxfn=1000;       // Maximum number of function evaluations is 1000
fmc.iprint=10;        // Current best estimate will be printed out every
                      // ten iterations
fmc.crit=.01;         // Convergence if derivatives are all smaller in
                      // magnitude than .01
fmc.min_improve=0.0;  // This convergence criterion won't be used
fmc.imax=0;           // This convergence criterion won't be used
gradient_structure gs;
while (fmc.ireturn >=0)
{
  fmc.fmin(f,x,g);
  if (fmc.ireturn > 0)
  {
    f=userfun(x,...) // The users function
    gradcalc(nvar,g);
  }
}
```

The function prototype for `fmin` is:

```
void fmm::fmin(double& f, independent_variables & x, dvector & g);
/*---------------------------------------------------------------------------*
 * Purpose: quasi-Newton numerical minimization                              *
 *  f       : current value of function to be minimized                      *
 *  x       : current values of the variables over which the minimization    *
 *            occurs; instance of class independent variables with elements  *
 *            from 1 to n                                                    *
 *  g       : current value of the gradient as returned by the function      *
 *            gradcalc( ... ); instance of class dvector with elements from  *
 *            1 to n                                                         *
 *---------------------------------------------------------------------------*/
```

The execution of `fmin` may be terminated by pressing the '`Q`' key on the keyboard. There may be a slight delay between the keypress and the termination of `fmin`.

It is often necessary to put bounds on the values that a parameter can take. For example if the expression $1/y$ is to be evaluated in a program an arithmetic error will occur if $y = 0$. Similarly e^x will probably produce an error if $x = 100000$.

The function `boundp` is supplied with AUTODIF to simplify the placing of bounds on parameters.

```
dvariable boundp( dvariable x, double fmin, double fmax, dvariable& fpen)
{
  dvariable t,y;
  t=fmin + (fmax-fmin)*(sin(x*1.570795)+1)/2;
  if (x < 0.00001)
  {
    fpen+=(x-0.00001)*(x-0.00001);
  }
  if (x > 0.9999)
  {
    fpen+=(x-0.9999)*(x-0.9999);
  }
  if (x < -1)
  {
    fpen+=1000*(x+1)*(x+1);
  }
  if (x > 1)
  {
    fpen+=1000*(x-1)*(x-1);
  }
  return(t);
}
```

The function `boundp` constrains the variable to lie between the values `fmin` and `fmax`. Both the values `fmin` and `fmax` can actually be attained so if you want a variable to be greater than 0 the code.

```
y=boundp(x,0.0,1.0,fpen);
```

will not do the job. Instead you can accomplish this in one of two ways. You can use a quadratic transformation if you really don't care how large x can become.

```
y=x*x;
```

You can make a decision about what is meant by "close to 0" for your application, in this case 0.00001.

```
y=boundp(x,0.00001,1.0,fpen);
```

The penalty function `fpen` is included in the `boundp` routine because otherwise the function would have a zero derivative at its lower and upper bounds. This causes the derivative with respect to the bounded parameter to be equal to 0. The result is that if the minimization routine is started with a parameter value already at the upper or lower bound then this parameter will never be changed. The penalty shifts the position of the critical point slightly within the bounds. Of course if the initial parameter estimate happens to coincide exactly with this critical point the derivative will be 0, but this is very unlikely.

If you neglect to add the penalty function to the function being minimized the minimization routine will still work. The only problem is that you run the risk of one of the bounded parameters getting "stuck" at its upper or lower bound.

The penalty should be added to the function which you wish to minimize. If you wish to maximize the function (in which case you are going to change the sign of the function before passing it to the function minimizer) then you should subtract the penalty. **If you get the sign of the penalty wrong the routines will not work properly.**

To begin the minimization you generally know the initial value of the constrained parameter `y`, and you need to find the value of `x` which will produce this value so that you can initialize the proper component of the vector of independent_variables with it. The function boundpin which is the inverse function of boundp will yield the desired value for x.

```
// double boundpin(double y, double fmin, double fmax)
x=boundpin(y,fmin,fmax);
```

It may happen that the derivatives being passed to fmin become incorrect for some values of the parameters. Two possible reasons for this are, an error in the internal derivative calculations of AUTODIF itself or, the use of a user's function which is not differentiable at the point in question. The former cause becomes increasingly less likely as experience with AUTODIF continues and any remaining bugs are removed. The latter can occur because when constructing a large complicated function it is always possible (and not uncommon in our experience) to add some non-differentiable components as an oversight.

If the function becomes nondifferentiable at some point or the derivative calculations are incorrect, the function minimizer will begin to experience difficulty in making progress. To help identify derivative errors or cases of non differentiability when they occur The AUTODIF system includes a interactive derivative checker which compares the value of the derivatives calculated by gradcalc() with finite difference approximations. The derivative checker can be invoked while the function minimizer is in progress by pressing the 'C' key on the keyboard.

The derivative checker operates in two modes. In the default mode, the derivative is checked for one independent variable at a time. In this mode you will be asked to supply the index of the variable for which you want to check the derivative. Alternatively, if you want to check all derivatives, enter 0 when prompted for the index of the variable to check. Next you will be asked to enter the step size to use for the finite difference approximation. Enter a number which will alter the value of the of the variables to be checked in the fifth or sixth significant digit. The derivative checker will then tabulate the values of the function, the variable being checked, the analytical derivative (from gradcalc()), the finite difference approximation, and the index of the variable being checked. The derivative checker may be interrupted by pressing the 'X' key on the keyboard. On completion, the derivative checker exits to the operating system.

For many models the finite approximations to the derivatives may change radically as the step size used in the derivative approximation is changed. How then can one be sure that the analytical derivatives are correct or incorrect. The following "rule of thumb" seems to work fairly well. First identify a gradient component which seems to be incorrect. Pick this component as the independent-variable whose derivative is to be checked. Evaluate the finite difference approximation for different step sizes, with the step sizes differing by a factor of about 3. Typical step sizes might be 1.e-4, 3.e-5, 1.e-5, 3.e-6, ... Now if the finite difference approximations stay almost the same for two consecutive step sizes, but are very different from the analytical approximations, this is a good indication that the analytical derivatives are incorrect.

It is wise to use the derivative checker frequently during the development of an application and after any change the the calculation of the objective function.

The derivative checker may also be invoked explicitly. Its function prototype is:

```
void derch(double & f, independent_variables & x, dvector & g,
           const int n, int & ireturn);
/*---------------------------------------------------------------------*
 * Purpose: compare analytical derivative calculation with finite difference *
 *          approximation                                              *
 * Input:                                                              *
 *  f       : current value of function                                *
 *  x       : current values of the parameters of the function; instance of  *
 *            class independent_variables with elements ranging from 1 to n  *
 *  g       : current value of the gradient as returned by the function      *
 *            gradcalc( ... ); instance of class dvector with elements from  *
 *            1 to n                                                   *
 *  n       : number of variables                                      *
 *  ireturn : return status; set ireturn = 3 prior to first call to derch()  *
 * Output:                                                             *
 *  ireturn : return status; ireturn = 4 or 5 on return from derch()   *
 *---------------------------------------------------------------------*/
```

The following code fragment will display the values of the function, the variable being checked, the analytical derivative (from `gradcalc()`), the finite difference approximation, and the index of the variable being checked.

```
#include <math.h>
#include <fvar.hpp>

// prototype for user supplied code to compute f as a function of x
void f_comp( double & f, independent_variables & x, ... );

void main()
{
  int n = ...;        // number of variables in function
  independent_variables x(n);
  set_x( ... );      //  user supplied code to set the values of x
  dvector g(1,n);
  gradient_structure gs;
  double f = 0;
  int derch_return = 3;

  f_comp(f, x, ... );   // first call to get value of function for current x
  while (derch_return >=3)
  {
    derch(f, x, g, n, derch_return);
    {
      f_comp(f, x, ... );    // subsequent calls for with values of x
                             // varied by the step size within derch()
      gradcalc(n, g);
    }
  }
}
```

Chapter 11

Statistical functions

The principal purpose of AUTODIF is for constructing computer programs that use mathematical derivatives. One such application is in nonlinear parameter estimation, that is the construction of nonlinear statistical models. It is convenient to use simulations which produce data with known statistical properties in order to analyze the behaviour of such models. To facilitate this process AUTODIF is furnished with a small number of statistical functions which can be applied to both constant and variable vector objects.

In this section, a uniformly distributed random number is assumed to have a uniform distribution on $[0,1]$. A normally distributed random number is assumed to have mean 0 and variance 1. A binomially distributed random number is assumed to have a parameter p where 1 is returned with probability p and 0 is returned with probability $1-p$. The `long int& n` is the random number seed specified by the user.

```
dvector::fill_randu(long int& n) // Fill a dvector from a uniform
                                 // distribution on 0,1
dvector::fill_randn(long int& n) // Fill a dvector from a standard normal
                                 // distribution
dvector::fill_randbi(int& n,double p) // Fill a dvector from a binomial
                         //distribution A 1 is returned with probability p
                         // A 0 is returned with probability 1-p
dvector::fill_multinomial(int& n,dvector p)// Fill a dvector from a multinomial
             //distribution The distribution is determined by the dvector p.
            // An i is returned with probability p(i) if the components of p
            // sum to 1. Otherwise p is normalized so that its components
            // sum to 1.
ivector::fill_multinomial(int& n,dvector p)// Fill a dvector from a multinomial
             //distribution The distribution is determined by the dvector p.
            // An i is returned with probability p(i) if the components of p
            // sum to 1. Otherwise p is normalized so that its components
            // sum to 1.
```

The `fill_multinomial` routine expects to receive a `dvector` of parameters $P = (p_1,\ldots,p_n)$. The routine fills the the elements of a `dvector` with integers whose values range from 1 to n where i occurs with probability p_i. If the components of P do not sum to 1 they will be normalized so that the components do sum to 1.

As an example of how to use these functions, consider the following code segment which generates random numbers from a normal distribution which is contaminated with a small proportion of large errors. This kind of distribution is useful for studying “robust” parameter estimation procedures. Robust estimation procedures are intended to be insensitive to such deviations from the main distribution.

Assume that the main distribution has mean 20 and standard deviation 2, while the contaminating distribution has mean 20 and standard deviation 6. We want the probability that the random numbers belong to the main distribution to be 0.90 and the probability that they belongs to the contaminating distribution to be 0.10. We shall generate a random vector of 100 such random numbers.

```
// ...
dvector rn(1,100);
{
  dvector w(1,100);
  rn.fill_randn(212); // fills rn with numbers drawn from a normal distribution
  w.fill_randbi(1521,0.90); // fills w with numbers drawn from a binomial
                            // distribution
  for (int i=1; i<=100;i++)
  {
    if (w(i)==1)  // This condition will occur with probability 0.90
    {
      rn(i)=2*rn(i)+20;  // standard deviation 2 and mean 20
    }
    else// This condition will occur with probability 0.10
    {
      rn(i)=6*rn(i)+20;  // standard deviation 6 and mean 20
    }
  }
} // Now the dvector w goes out of scope and the memory it used is released
// ...
```

This example extends the previous example to a mixture of more than two distributions. Two interesting aspects of this example are the use of the multinomial fill of an `ivector` object and the use of the resulting `ivector` object in a general selection of the form `dvector(ivector)`.

```
int nsample;  // nsample is the sample size to be generated
int n;        // n is the number of groups in the mixture
dvector p(1,ngroups); // p contains the mixture proportions;
dvector mean(1,ngroups); // mean contains the mean for each group
dvector sd(1,ngroups); // sd contains the standard deviations for each group
dvector sample(1,nsample); // sample will contain the simulated data
ivector choose(1,nsample); // this will determine which group in the mixture
                           // each observation came from
//...
// somehow get the mixture proportions into p, the means into mean,
// and the standard deviations into sd
// ...
choose.fill_multinomial(1011,p);

sample.fill_randn(211); // fill the sample with standard normal deviates
sample=elem_prod(sd(choose),sample)+mean(choose); //This
            // creates the mixture of normal distributions
```

Notice that the i'th element of `choose` will be equal to j with probability p_j so that each element in sample will have a probability p_j of having mean(j) as its mean and sd(j) as its standard deviation. This means that each element of `sample` will have a probability p_j of belonging to the j'th group of normal distributions. It follows that the elements of `sample` have the desired mixture distribution.

The functions

```
number mean(vector_object);
number std_dev(vector_object);
```

return the mean and standard deviation of a `dvector` or `dvar_vector`.

Chapter 12

Robust Nonlinear Regression

Many authors have indicated that the usual method of least squares estimation of parameters is too sensitive to the existence of a small number of large errors, so-called outliers, in the data being analyzed. Although an extensive theory of robust estimators which are not so sensitive to such outliers has developed, many people still use standard statistical packages based on least squares estimation for statistical problems such as multiple regression. With the AUTODIF system it is a simple matter to use superior robust regression estimation procedures on both linear and nonlinear statistical models. We should emphasize that we are not experts in the field of robust regression and we make no claims of optimality or near optimality for these methods. The methods in this chapter seem to work well and they have been designed so that they can be easily extended to large nonlinear regression problems with thousands of observations and at least hundreds and perhaps thousands of parameters. (In 16 bit DOS versions of AUTODIF the 64K barrier may limit the size of the problem which can be considered).

Our purpose in this chapter is to present an estimation procedure which performs almost as well as the usual least squares estimator when the model errors are normally distributed and which performs much better than least squares when the model errors contain some large outliers. With such an estimator you don't have to worry about whether the use of robust techniques is justified. If your model errors are exactly normally distributed you have lost very little and if they are not normally distributed you may have gained a lot.

The reader who is not interested in the derivation of the formulas in this section can skip to section 12.2 where the methods for using the routines are discussed. The necessary functions are supplied with the AUTODIF libraries.

Assume that we have n data points, Y_i, $i = 1, \ldots, n$. The Y_i are assumed to have been produced by the process

$$Y_i = f(\boldsymbol{x}_i; \Theta) + \epsilon_i \tag{12.0}$$

where for each i, $\boldsymbol{x}_i$ is a known vector, Θ is an unknown vector of parameters, and the ϵ_i are random variables. We want to get estimates for the parameters Θ. Some common special cases are:

Multiple regression: for each i, $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ir})$ is an r-vector and $\Theta = (\theta_0, \theta_1, \ldots, \theta_r)$ is an $r+1$-vector. The functional relationship is

$$f(\boldsymbol{x}_i; \Theta) = \theta_0 + x_{i1}\theta_1 + x_{i2}\theta_2 + \ldots + x_{ir}\theta_r$$

Polynomial regression: for each i, $\boldsymbol{x}_i = (x_i)$ is a 1-vector and $\Theta = (\theta_0, \theta_1, \ldots, \theta_r)$ is an $r+1$-vector. The functional relationship is

$$f(\boldsymbol{x}_i; \Theta) = \theta_0 + x_i\theta_1 + x_i^2\theta_2 + \ldots + x_i^r\theta_r$$

A nonlinear model: For a simple nonlinear model we shall condsider the case where $\boldsymbol{x}_i$ is a 1-dimensional vector and Θ is a 1-dimensional vector whose single component is denoted by b. The functional relationship is

$$f(\boldsymbol{x}_i; b) = bx_i + b^2 \tag{12.1}$$

The form of the estimation procedure which should be used to estimate the parameter vector Θ depends on the assumptions which are made about the nature of the random variables ϵ_i. Assume that the ϵ_i are independent identically distributed random variables with a probability density function $\phi(u; \Lambda)$ where Λ is a vector of parameters whose components may be known or unknown.

The joint probability density function for the $(Y_1, \ldots, Y_n)$ is given by

$$\textstyle\prod_{i=1}^n \phi(Y_i - f(\boldsymbol{x}_i; \Theta); \Lambda) \tag{12.2}$$

Taking the logarithm of (12.2) gives the expression

$$\textstyle\sum_{i=1}^n \log\Big(\phi(Y_i - f(\boldsymbol{x}_i; \Theta); \Lambda)\Big) \tag{12.3}$$

Considered as a function of the parameters Θ, and Λ expression (12.3) is known as the log-likelihood function. The maximum likelihood estimates for the parameters Θ and Λ is the value of parameters which maximizes the likelihood, or what is equivalent, the log-likelihood function.

If we assume that the ϵ_i are normally distributed with mean 0 and standard deviation σ then

$$\phi(u; \Lambda) = \frac{1}{\sqrt{2\pi}\sigma} \exp\big(-\frac{u^2}{2\sigma^2}\big)$$

and the log-likelihood function becomes

$$-n\log(\sigma) - \tfrac{1}{2\sigma^2}\textstyle\sum_{i=1}^n \left(Y_i - f(\boldsymbol{x}_i; \Theta)\right)^2 \tag{12.4}$$

In this case $\Lambda = (\sigma)$ is a 1 dimensional vector.

The maximum likelihood estimates estimates $\widehat{\Theta}$ for the parameters Θ which maximize (12.4) are the same as the those values of the parameters which minimize

$$\textstyle\sum_{i=1}^{n} \big(Y_i - f(\boldsymbol{x}_i;\Theta)\big)^2 \tag{12.5}$$

Since these parameter estimates minimize the squared difference between the predicted and observed values they are known as the least-squares estimates. For normally distributed model errors the maximum likelihood estimates and the least squares estimates coincide. Since the expression (12.5) does not depend on σ, The maximum-likelihood estimate for Θ does not depend on the estimate for σ. This is a special property shared by the normal distribution and some other distributions. It is does not hold in general. The maximum likelihood estimate $\hat{\sigma}$ for the standard deviation σ is given by

$$\hat{\sigma} = \sqrt{\tfrac{1}{n}\textstyle\sum_{i=1}^{n} \big(Y_i - f(\boldsymbol{x}_i;\Theta)\big)^2} \tag{12.6}$$

The reason that the normal least squares estimation is so sensitive to outliers is that the normal distribution has a small "tail", so that the probability of occurrence of an event drops off very quickly as one moves away from the mean a distance of a few standard deviations. One approach to develop more robust estimation methods is to use distributions with fatter tails. A distribution with a fatter tail than the normal distribution is the double exponential distribution whose probability density function (assuming the mean is 0) is given by

$$\tfrac{1}{2\sigma}\exp|-\tfrac{u}{\sigma}| \tag{12.7}$$

The corresponding log-likelihood function is

$$-n\log(\sigma) - \tfrac{1}{\sigma}\textstyle\sum_{i=1}^{n} |Y_i - f(\boldsymbol{x}_i;\Theta)| \tag{12.8}$$

The values of the parameters Θ which maximize (12.8) are the same as the values which minimize the sum

$$\textstyle\sum_{i=1}^{n} |Y_i - f(\boldsymbol{x}_i;\Theta)| \tag{12.9}$$

Since (12.9) is the total absolute deviation between the predicted and observed values these estimates are known as the minimum total absolute deviation estimates for the parameters.

While the minimum total absolute deviation estimates are fairly insensitive to a small number of large model errors, they do not perform very well when the model errors are normally distributed. To develop a maximum likelihood estimation procedure which is less sensitive to outliers, but which also performs well when the model errors are normally distributed, we have employed a distribution which is a mixture of a normal distribution and another distribution with a fatter tail. The probability density functions of the fat-tailed distribution is given by

$$\tfrac{\sqrt{2}}{\pi\sigma e}\Big\{1+\tfrac{x^4}{(\sigma e)^4}\Big\}^{-1} \tag{12.10}$$

The extra parameter e has been added to the log-likelihood function (12.10) to adjust the “spread” of the fat-tailed distribution. The value of e has been set equal to 3. A mixture of two random variables with mixing proportion p is formed by choosing one of the random variables with probability p and the other random variable with probability $1-p$. If the probability density functions of the components of the mixture are $\psi(u)$ and $\chi(u)$ then the probability density function of the mixture is $p\psi(u)+(1-p)\chi(u)$. We shall assume that the model errors ϵ_i are a mixture of the distribution (12.10) with probability 0.05 and a normal distribution with probability 0.95. The quantity 0.05 can be interpreted as the proportion of observations contaminated by large errors.

The corresponding log-likelihood function for the observations is

$$-n\ln(\sigma)+\textstyle\sum_{i=1}^{n}\log\Big[(1-p)\exp\Big\{-\tfrac{\big(Y_i-f(\boldsymbol{x}_i;\Theta)\big)^2}{2\sigma^2}\Big\}+\tfrac{2p}{\sqrt{\pi}e}\Big\{1+\tfrac{\big(Y_i-f(\boldsymbol{x}_i;\Theta)\big)^4}{(e\sigma)^4}\Big\}^{-1}\Big] \tag{12.11}$$

The parameter estimates $\widetilde{\Theta}$ and $\tilde{\sigma}$ for Θ and σ obtained by maximizing (12.5) will be referred to as the robust-mixture estimator for the parameters. For the robust-mixture estimator the estimate for Θ is not independent of the estimate for σ, so that to estimate the parameters the entire expression (12.11) must be maximized for Θ and σ simultaneously.

While it is true that the robust parameter estimates are the values of the parameters which maximize (12.11) it is possible to reformulate the problem slightly so that it is posed in terms of parameters which are more easily estimated. Such reparameterizations are often useful in nonlinear optimization. Consider the log-likelihood function (12.11). If p is set equal to 0 the log-likelihood reduces to the normal case, expression, (12.4) so that the value of the parameter σ which would maximize (12.11) when $p=0$ is $\hat{\sigma}$ given by (12.6).

Define a new parameter α by $\alpha=\sigma/\hat{\sigma}$ so that $\sigma=\alpha\hat{\sigma}$. Replace the parameter σ in (12.11) by the parameter $\alpha\hat{\sigma}$. The log-likelihood function now takes the form

$$n\ln(\alpha\hat{\sigma})-\sum_{i=1}^{n}\log\Big[(1-p)\exp\Big\{-\frac{\big(Y_i-f(\boldsymbol{x}_i;\Theta)\big)^2}{2(\alpha\hat{\sigma})^2}\Big\}+\frac{2p}{\sqrt{\pi}e}\Big\{1+\frac{\big(Y_i-f(\boldsymbol{x}_i;\Theta)\big)^4}{(e\alpha\hat{\sigma})^4}\Big\}^{-1}\Big] \tag{12.12}$$

The parameter α can be interpreted as an estimate of how much of the residuals in the model fit are contributed by the large outliers generated by the contaminating distribution. If α is close to 1 most of the contribution to the residuals comes from the “ordinary” errors associated with the normal distribution . If α is close to 0 most of the contribution to the residuals comes from the large outliers associated with the fat-tailed distribution. We expect that the maximum likelihood estimate $\tilde{\alpha}$ for α will lie between 0 and 1. The parameter α can also be interpreted as a “cutoff” switch. As α is made smaller the influence of the larger residuals on the parameter estimates will be reduced. This is much like looking at the data and removing the data points which fit the model badly.

The code for the log-likelihood function 12.12 can be written down very concisely using AUTODIF's vector operations. suppose that the observations Y_i are contained in a `dvector obs` and the predicted values $f(\boldsymbol{x}_i;\Theta)$ are contained in a `dvar_vector pred`. The vector of residuals $Y_i - f(\boldsymbol{x}_i;\Theta)$ and the vector of squared residuals can be calculated in two statments

```
dvar_vector diff = obs-pred;     // These are the residuals
dvar_vector diff2 = pow(diff,2); // These are the squared residuals
v_hat = mean(diff2);
```

and the estimate of the variance is given by the mean of the vector of squared residuals. If `b` $= 2p/\sqrt{\pi}e$ and `a2` $= 2\alpha$ the code for 12.12 can be written as

```
dvariable log_likelihood = 0.5*diff.size()*log(a2*v_hat)
  -sum(log((1.-pcon)*exp(-diff2/(2*a2*v_hat))
  + b/(1.+pow(diff2/(a2*v_hat),2))));
```

While the combination of scalars and vectors in statements like `1.+v` can seem a bit strange at first one quickly gets used ot it. The main thing to consider is that some combinations of operations are not associative so that it may be necessary to include parentheses to ensure that the operations are carried out in the correct order. As is often the case, using vector operations instead of writing the code in terms of the components of the vector objects will greatly reduce the amount of temporary storage required.

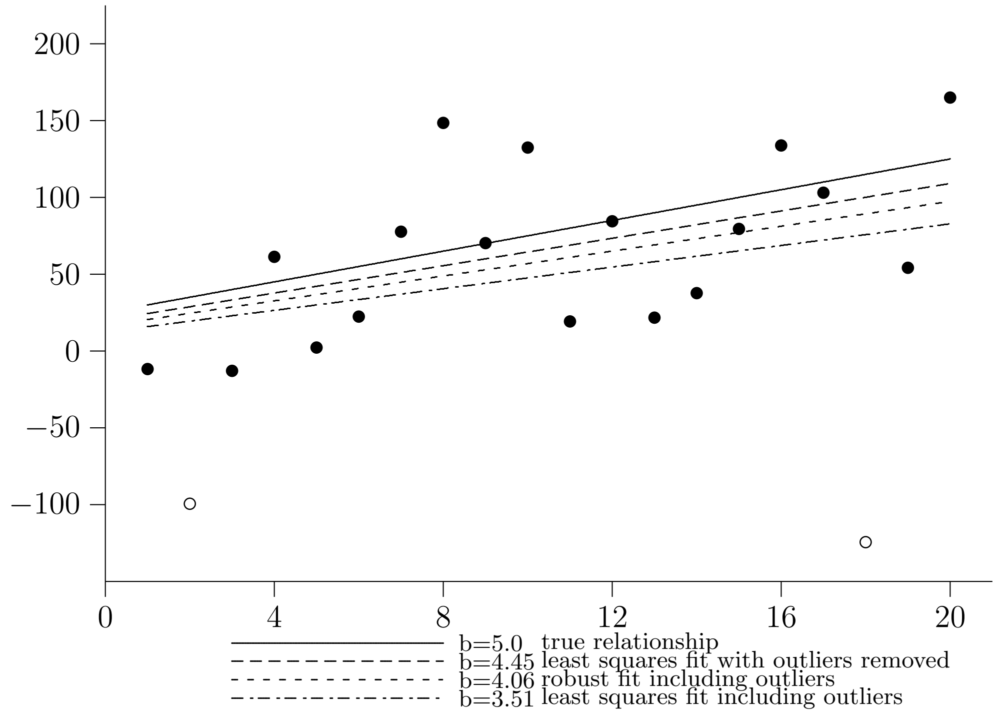
As the examples below illustrate the parameter α is often not well determined by the data. Rather than estimating this parameter dirctly we feel that the best way to do robust regression is probably to use a fixed value for α. We recommend the value 0.7 for fitting the model although we have no strong reason for preferring this particular value.

The AUTODIF system permits the user to obtain these robust regression estimates in a simple fashion. Suppose that the observations Y_i are contained in a `dvector OBS_Y` and that the values predicted by the model for these observation have been computed and are contained in a `dvar_array PRED_Y`. To compute the value of the log-likelihood function (12.5) with a fixed value of α the routine `robust_regression` is used. The function prototype for this routine is

```
dvariable robust_regression(dvector& OBS_Y,dvar_vector& PRED_Y,double& alpha);
```

The function `robust_regression` returns minus the value of the log-likelihood function. (The sign has been changed for the function minimizer.)

Figure 12.1



To estimate the parameter α as well, the parameter must be passed as a `dvariable`. The function prototype is

```
dvariable robust_regression(dvector& OBS_Y,dvar_vector& PRED_Y,dvariable& alpha);
```

For the routine `robust_regression` to work properly when the parameter α is being estimated it is necessary to put bounds on the values that α can assume. This is done by using the `boundp` function supplied with AUTODIF. The parameter α must be restricted so that $0 < a <= 1$. (Actually the upper bound is not necessary, but it will make the routine run a bit more efficiently.) The code

```
  a=boundp(x(i),.001,1.0,zpen);
```

restricts the value of α to lie within 0.001 and 1.0. To find the initial value which will give a desired initial value for α, the inverse function `boundpin` can be used. To obtain the initial value which gives a value of 0.7 for α and put the initial value in the ith component of the vector of independent variables `x` the code

```
  x(i)=boundpin(0.7,001,1.00);
```

would be employed.

We shall fit a simple 1 parameter model using the robust regression routine `robust_regression` which estimates the value of the parameter α. The form of the model is

$$Y_i = bZ_i + b^2$$

where the true value of b is 5. The data for this example were produced by simulating 18 real data points with normally distributed errors with mean 0 and standard deviation 40, and then adding two outliers which were 3.1 standard deviations and 6 standard deviations removed from the true value. Of course such large outliers could be identified by eye in this simple example. For large nonlinear models, however, there is often no simple graphical representation of the fit which can enable such a simple inspection of the residuals. A $\circ$ was used to mark the outliers.

The results of the fitting procedure are shown in figure 12.1. The least squares fit with the outliers removed (estimated $b = 4.45$) is the closest to the true relationship. It represents the best estimates which can be obtained if the outliers are identified with certainty. The robust fit including the outliers (estimated $b = 4.06$) did condiderably better than the least squares fit including the outliers (estimated $b = 3.51$).

For the robust fit the estimate of $\hat{\sigma}$ is 70.1 while the estimate of $\tilde{\sigma}$ is 55.6 so that the estimate of α is 0.79. Since the real value of σ was 40.0 the robust regression routine has underestimated the contribution to the residuals made by the outliers.

The AUTODIF code for this model is given below: `// file: robust.cpp`

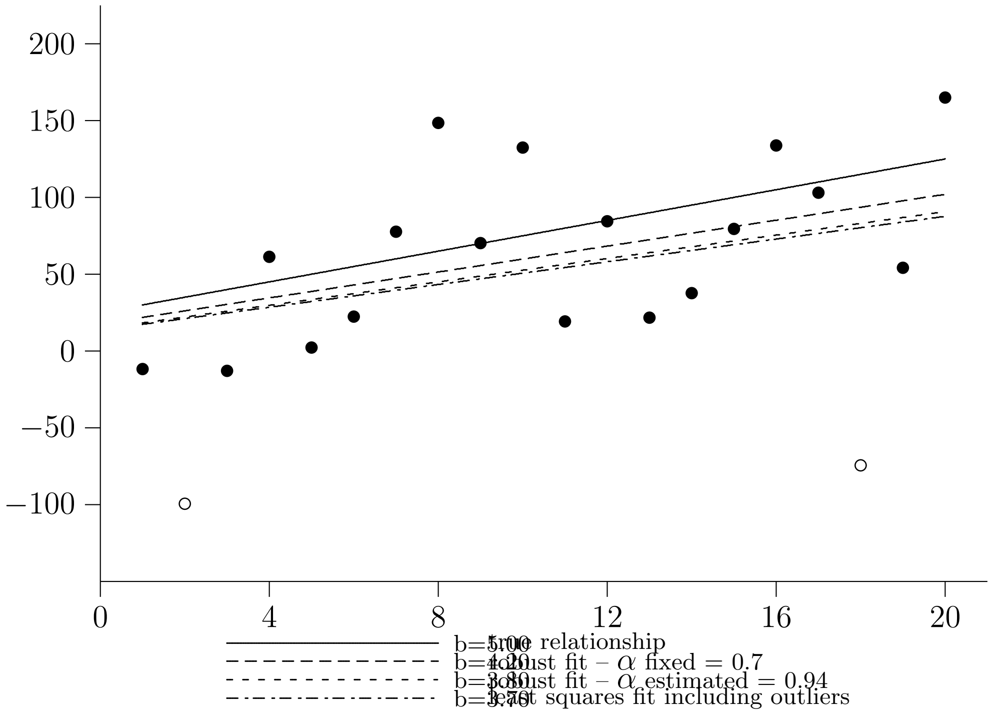
```
#include <fvar.hpp>

// function prototype
double fcomp(dvector& OBS_Y,dvector& OBS_Z,dvar_vector& x);

#ifdef __BCPLUSPLUS__
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  long _stack = 20000;
#endif
double alpha;
void main()
{
  ifstream infile("robust.dat"); // robust.dat contains the observed data
  if(!infile)  // Check for I/O error
  {
    cerr << "Error trying to open the file robust.dat\n";
    exit(1);
  }
  int nobs;  // nobs is the number of observations
  infile >> nobs; // read in the number of observations
  if(!infile)  // Check for I/O error
  {
    cerr << "Error reading number of observations from the file robust.dat\n";
    exit(1);
  }
  dvector Y(1,nobs);
  dvector Z(1,nobs);
  // The Data are in the file in the form one Y_i Z_i pair per line
  {  // Limit the scope of the matrix tmp
     dmatrix tmp(1,nobs,1,2);
     infile >> tmp; // Read the  Y_i Z_i pairs into tmp
     if(!infile)  // Check for I/O error
     {
       cerr << "Error reading data in from the file robust.dat\n";
       exit(1);
     }
     Z=extract_column(tmp,1);  // Put the first column of tmp into Z
     Y=extract_column(tmp,2);  // Put the second column of tmp into Y
  } // Now tmp goes out of scope
  infile.close();  // Close the input file
  double b=1.0; // This is the model parameter ... use 1.0 as initial value
  alpha=0.7; // alpha is not used in the first stage of the minimization
  double f;
  gradient_structure gs;
  {
    int nvar=1; // There is 1 parameter, b
    independent_variables x(1,nvar);
    dvector g(1,nvar);
    x[1]=b; // Put the model parameters in the x vector
    fmm fmc(nvar);
    fmc.iprint=1;
    fmc.crit=1.e-6;
    while (fmc.ireturn >=0)
    {
      fmc.fmin(f,x,g);
      if (fmc.ireturn > 0)
      {
        {
          f=fcomp(Y,Z,dvar_vector(x));
        }
        gradcalc(nvar,g);
      }
    }
    cout << " The estimate for b = " << x(1) << "\n";
  }
}
```

```
// file: fcomp_r.cpp

#include <fvar.hpp>

double fcomp(dvector& Y,dvector& Z,dvar_vector& x)
{
  dvariable b;
  b=x[1]; //Put the x vector into the model parameter
  dvar_vector PRED_Y=b*Z+b*b; //calculate the predicted response
  double alpha=0.7;
  dvariable tmp =robust_regression(Y,PRED_Y,alpha);
  return(value(tmp));
}
```

```
#include <fvar.hpp>
dvariable robust_regression(dvector& obs, dvar_vector& pred, dvariable& a)
{
  if (obs.indexmin() != pred.indexmin() || obs.indexmax() != pred.indexmax() )
  {
    cerr << "Index limits on observed vector are not equal to the Index\n  limits on the
predicted vector in robust_reg_likelihood function\n";
  }
  RETURN_ARRAYS_INCREMENT(); //Need this statement because the function
     //returns a variable type
  dvariable v_hat;
  double width=3.0;
  double pcon=0.05;
  double width2=width*width;
  dvariable a2;
  a2=a*a;
  dvar_vector diff = obs-pred;     // These are the residuals
  dvar_vector diff2 = pow(diff,2); // These are the squared residuals
  v_hat = mean(diff2)+1.e-80;
  double b=2.*pcon/(width*sqrt(3.14159));  // This is the weight for the
                                           // "robustifying" term
  // Use vector calculations to do the entire log-likelihood function in
  // two statements
  dvariable log_likelihood = -sum(log((1.-pcon)*exp(-diff2/(a2*v_hat))
    + b/(1.+pow(diff2/(width2*a2*v_hat),2))));
  log_likelihood += 0.5*diff.size()*log(a2*v_hat);
  RETURN_ARRAYS_DECREMENT(); // Need this to decrement the stack increment
                             // caused by RETURN_ARRAYS_INCREMENT();
  return(log_likelihood);
}
```

It is important to be able to specify the ratio $\tilde{\sigma}/\hat{\sigma}$ because this parameter is often not well determined by the data. In the previous example the robust estimation procedure has somewhat underestimated the contribution to the overall variance of the residuals made by the outliers. Now consider the analysis for the same data as above with one change. The value of the second outlier has been set equal to -75.0 rather than -125.0. Although this represents an improvement in the quality of the data set, the parameter estimates (estimated $b = 3.80$) for the robust-mixture estimation procedure when the parameter α is estimated are actually worse than before. The reason for this is that the estimation of the ratio $\alpha = \tilde{\sigma}/\hat{\sigma}$ has increased to 0.94 which means that the robust estimation procedure no longer recongnizes that there are outliers in the data. the robust-mixture estimates to the data for which the ratio $\tilde{\sigma}/\hat{\sigma}$ has been fixed at 0.7 (estimated $b = 4.20$) is included. This is considerably better than the least squares fit (estimated $b = 3.70$).

Figure 12.2



We suspect that a little “robustness” is much better than none at all and that overall best performance of the model may be obtained by simply fixing the value for the ratio $\tilde{\sigma}/\hat{\sigma}$ at some nominal value such as 0.7.

While the above example is indicative of the superiority of the robust-mixture estimator, it only indicates how the estimators perform in one special case. From a statistical point of view one is interested in the average performance of the estimators for many different realizations of the data. This section describes the results of such an investigation. The data were fit by the least squares estimator, the minimum total absolute deviation estimator and the robust-mixture estimator (with fixed $a = 0.7$).

For each simulation a set of 30 pairs (Y_i, z_i) were generated where $z_i = i$, for $i = 1, \ldots, 30$ and

$$Y_i = bz_i + b^2 + \epsilon_i$$

The parameter b was set equal to 5. For the case of normally distributed model errors, the ϵ_i are normally distributed random variables with standard deviation 40. For model errors exhibiting moderate deviations from normality the ϵ_i are a mixture normally distributed random variables with standard deviation 40 with probability 0.90 and normally distributed random variables with standard deviation 120 with probability 0.10. For model errors exhibiting large deviations from normality ϵ_i were a mixture of normally distributed random variables with standard deviation 40 with probability 0.90 and normally distributed random variables with standard deviation 200 with probability 0.10.

Five hundred replications were made for each of the error distributions. For all the simulations the value of ratio α was kept fixed at 0.7 for the robust-mixture estimator.

Now it is known that for normally distributed errors, the usual least squares estimates are the minimum variance unbiased estimates of the model parameters. It follows that if the robust estimates are unbiased they must have greater variance than the least squares estimates. This greater variance is the penalty we must pay for using robust estimates in the case where they are not necessary. For the problem studied here, however, the standard deviation of the robust estimates was only about 8% greater than the least squares estimates for the case of normally distributed errors, while the robust-mixture estimator had a standard deviation about 20% smaller than that of the least squares estimates for moderate deviations from normality and about 35% smaller for large deviations from normality.

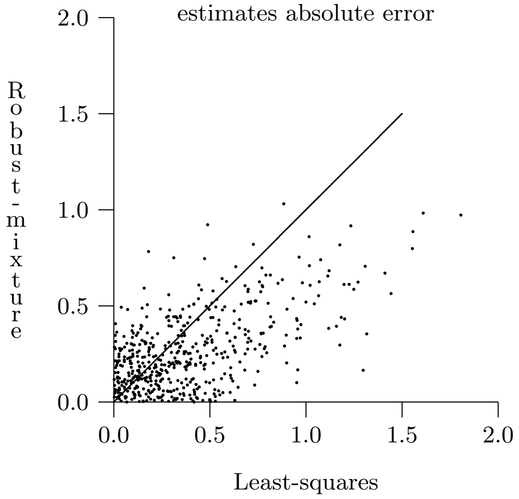
Table 12.1 Comparison of the least squares (LS), Minimum total absolute deviation (MD), and robust-mixture (RM) estimates for b in the model $Y = bz + b^2$. The true value was $b = 5.0$

	Normal errors			Moderate deviations			Large deviations		
Mean	LS	RM	MD	LS	RM	MD	LS	RM	MD
Mean	5.01	5.01	5.00	4.99	5.01	5.01	4.98	4.99	5.01
Std Error	0.012	0.013	0.015	0.016	0.014	0.016	0.023	0.015	0.016
Std Dev	0.26	0.28	0.33	0.36	0.30	0.36	0.51	0.33	0.36
% improvement		-0.077	-0.27		20.0	0.00		35.3	29.4

All three estimators appear to be unbiased. The standard deviation of the least squares estimates almost doubles as we go from the normal errors to errors with large deviations from normality. The robust-mixture estimator exhibits much greater stability. The minimum total absolute deviation estimator performs almost as well as the robust-mixture estimator when there are large outliers present, but does not perform very well for normally distributed errors or for medium-sized outliers present.

Another way to compare the performance of the robust-mixture estimator over the least-squares estimator is illustrated by Figure 12.2. The absolute value of the difference between the robust-mixture estimator for b and the actual value of b is plotted against the absolute value of the difference between the least-squares estimator for b and the actual value of b. For points which lie below the diagonal line the robust-mixture estimator has smaller absolute error than the least-squares estimator. The plot is for the large-deviations set of simulations.

Fig. 12.3 Plot of Robust-mixture estimates absolute error vs Least-squares estimates absolute error



While the errors in the data in the previous examples exhibited moderate and large deviations from normality, the data were not really "bad". A much more difficult problem for parameter estimation is created if all the large errors are concentrated on one side of the true line. To compare the estimation methods for this kind of "bad" data we generated a data set where the majority (30 data points) of the data are distributed around the line $y = x$, with the outliers (20 data points) distributed around the point (6,2). The data were fit with the two parameter linear model

$$Y = a + bx$$

where the true values were $a = 0.0$ and $b = 1.0$.

Various fits to the data are shown in figure 12.3. When the value of α is set equal to 0.1 the estimation procedure completely ignores the 20 outliers. We anticipate that varying the value of α and noting whether the residuals suddenly shift could be a useful way to detect these types of outliers in large nonlinear models. Of course in a large model it is not always possible to view the results in this simple way. One way to examine how the vector of residuals is changing is to calculate the euclidean distance between the vectors. If the residuals are changing slowly the distance between the residuals should be small. A sudden shift in the residuals should manifest itself in a large distance between the vectors.

Figure 12.3

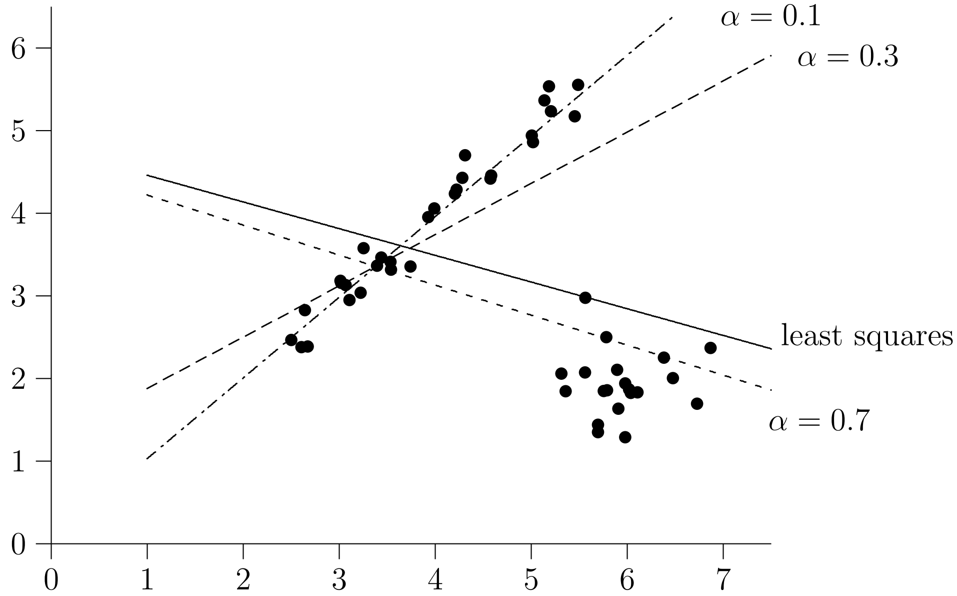


Table 12.2 shows the distance between the vectors of residuals for different values of α. It is clear that there are two groups, one group for values $0.9 > \alpha > 0.4$ and another group for $0.3 > \alpha > 0.1$. Constructing such a table could be a possible way to search for "interesting" structure in the outliers in a large nonlinear model.

Table 12.2 Euclidean distance between the vectors of residuals for the robust-mixture estimator for different values of α

α	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
0.9	0.0								
0.8	0.3	0.0							
0.7	1.0	0.6	0.0						
0.6	1.9	1.6	0.9	0.0					
0.5	2.7	2.4	1.7	0.7	0.0				
0.4	3.4	3.1	2.4	1.5	0.7	0.0			
0.3	11.6	11.9	12.5	13.4	14.1	14.7	0.0		
0.2	14.0	14.3	14.9	15.8	16.5	17.1	2.4	0.0	
0.1	15.9	16.2	16.9	17.7	18.4	19.0	4.3	1.9	0.0

Chapter 13

Problems with more than one dependent variable

Till now all the problems considered have had only one dependent variable. Many problems of interest such as the problem of solving a system of nonlinear equations require the computation of derivatives with respect to more than one dependent variable.

The operator << has been overloaded to implement more than one dependent variable. The user simply writes something like

```
u << -exp(x(1))+y; // u is a dependent variable
```

where u is a dvariable and u will become a dependent variable for which derivatives can be calculated. The operator << acts like = with the additional property that u is made a dependent variable. Of course it is possible to use u in later calculations such as

```
u << -exp(x(1))+y; // u is a dependent variable
// ...
v << u*u; // v is a dependent variable
```

The process of declaring depdendent variables in this way is global, that is, it does not have to be doen in one function. the dependent variables are ordered in the order in which they are encounterd when the code is executed.

The default maximum number of dedendent variables (which is at least 10) depends on the implemntation of AUTODIF which you are using. If desired it can be increased by using the static member function

```
gradient_structure::set_NUM_DEPENDENT_VARIABLES(int i);
```

For example to set the maximum number of depdendent variables to 25 you would put the line

```
gradient_structure::set_NUM_DEPENDENT_VARIABLES(25);
// ...  must occur before the gradient_structure declaration
gradient_structure gs;
```

into your code. The derivatives are calculated by calling the function `jacobcalc(int& nvar,dmatrix& jac)`. The derivatives are put into the `dmatrix jac`. The minimum valid row index of `jac` must be 1 while the maximum valid row index must be greater than or equal to the number of dependent variables. Each row of `jac` must have a minimum valid index of 1 and a maximum valid index greater than or equal to `nvar` where `nvar` is the number of independent variables.

Suppose that you wish to solve the nonlinear system of equations

$$\begin{array}{ll} f_1(x_1,\ldots,x_n) & = 0 \\ f_2(x_1,\ldots,x_n) & = 0 \\ f_2(x_1,\ldots,x_n) & = 0 \\ \ldots & \\ f_n(x_1,\ldots,x_n) & = 0 \end{array} \tag{13.1}$$

In general this is a very difficult problem. However, under suitable regularity conditions, if you can find an initial estimate for $(x_1,\ldots,x_n)$ which is close enough then the iterative Newton-Raphson technique will efficiently converge to the solution. Let X denote the vector $(x_1,\ldots,x_n)$

$$J(X) = \frac{\partial f_i(X)}{\partial x_j}$$

be the Jacobian matrix. Then to first order

$$f(X+\delta) = f(X) + J(X)\delta(13.2)$$

where $f(X)$ denotes the vector $f_1(X),\ldots,f_n(X)$ and $J(X)\delta$ denotes the multiplication of the vector δ by the matrix $J(X)$. Equating (13.2) to zero and solving for δ yields

$$\delta = -J(X)^{-1}f(X)(13.3)$$

Now we proceed iteratively. If $X^{(r)}$ is the current estimate for X let

$$X^{(r+1)} = X^{(r)} + \delta$$

be the next estimate where δ is defined by (13.3).

This example shows the Newton-Raphson technique for two equations. //file:

```
newt.cpp
#include <fvar.hpp>
//  function prototypes for user's functions
dvector function(dvar_vector& x,int nvar);
#ifdef __BCPLUSPLUS__
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  long _stack = 20000;
#endif
void main()
{
  int nvar=2;
  independent_variables x(1,nvar);
  dvector f(1,nvar);
  dmatrix jacobian(1,nvar,1,nvar);
  gradient_structure gs;
  x[1]=1;
  x[2]=2;
  do      // Iterate until the norm of the dvector f is < 1.e-12
  {
    f=function(x,nvar);
    jacobcalc(nvar,jacobian); // calculate the derivatives for the
                         // dependent variables
    cout << " The x vector        = " << x << endl
         << " The function values = " << f << endl << endl;
    x-=inv(jacobian)*f; // This is the Newton-Raphson step
  }
  while (norm(f)>1.e-12);
    // print out some final statistics
    cout << " Final x vector values       = " << x << endl
         << " Final function values       = " << f << endl
         << " Final jacobian matrix value = " << jacobian(1) << endl
         << "                               " << jacobian(2) << endl;
    cout << " The angle between the rows of the jacobian = "
         << acos(jacobian(1)*jacobian(2)
              /(norm(jacobian(1))*norm(jacobian(2))))
              *180/3.14159 << " degrees" << endl;
    cout << " Final inverse jacobian matrix value = "
         << inv(jacobian)(1) << endl
         << "                                       "
         << inv(jacobian)(2) << endl;
}

dvector function(dvar_vector& x,int nvar)
{
  double offset=.0001;  // try changing offset a bit to see how
                       // unstable the roots are
  dvar_vector f(1,nvar);
  dvariable z,tmp;
  f[1] << 100*x[1]*x[1]+x[2]*x[2]-1.; // << sets the value of the first
                          //  dependent variable
  // now use the first dependent variable in calculation for the
  // second dependent variable
  f[2] << f[1]+offset-.01*pow(x(2)*x(2),2)+.0001*pow(x(1)*x(1),2);
  return value(f);
}
```

We have chosen a rather “degenerate” example where the Jacobian matrix is almost singular at the solution. The two functions are

$$
\begin{aligned}
f_1(x_1,x_2) &= 100x_1^2 + x_2^2 - 1 \\
f_2(x_1,x_2) &= 100x_1^2 + x_2^2 - 1 + .0001 - .0001x_1^4 + .01x_2^2
\end{aligned}
\tag{13.2}
$$

The calculation of the jacobian and the Newton-Raphson “update” step can be accomplished with two lines of code.

```
jacobcalc(nvar,jacobian)
x-=inv(jacobian)*f
```

The iteration continues until the norm of the `dvector f` is less than `1.e-12`.

The only variable types occur in the user’s function `function`. All the arithmetic for the Newton-Raphson update is done with vectors and matrices which are of constant type. The `value` function is used to turn the `dvar_vector f` in `function` into a `dvector` object so that it can be returned to the calling routine.

Chapter 14

A Complete Nonlinear Parameter Estimation Program

A major difference between linear and nonlinear statistical modelling is the inherent instability of nonlinear models. It seems to be an empirical fact that for any reasonably complicated model if you simply write correct naive computer code for estimating the parameters, the program will not work. By "work" we mean get the information which you want out of the data you are trying to analyze. In general there are two reasons for this behaviour.

1. The function being calculated is numerically unstable so that a zero divide or a floating point overflow will occur, for example calculate $1/x$ when x has the value 0 or calculate $\exp(x)$ when x has the value 90000.

2. The "correct" values of the parameter estimates occur at a local minimum of the objective function, not at a global minimum.

The problem we consider in this chapter exhibits both kinds of pathology mentioned above. We will describe the techniques which are necessary for producing a stable, controllable nonlinear parameter estimation routine for solving this problem. The code for this example is included on the AUTODIF distribution disk.

Consider a set of m random variables X_j, indexed by j, with probability density functions $F_j(x)$. A new random variable Y can be created from the X_j as follows. A realization of Y is produced by first picking one of the X_j at random with probability p_j and then taking a realization of the resulting X_j. The random variable Y is a finite mixture of the X_j with mixing proportions p_j. The probability density function for Y_j is given by $\sum_{j=1}^{m} p_j F_j(x)$. The particular finite mixture which we shall address is a mixture of normal distributions. Effective techniques for estimating parameters for this problem were first considered by Hasselblad (1966).

Assume that X_j is normally distributed with mean μ_j and standard deviation σ_j. The probability density function for X_j is given by

$$\frac{1}{\sqrt{2\pi}\sigma_j}\exp\Big(\frac{-(x-\mu_j)^2}{2\sigma_j^2}\Big)$$

while the probability density function for the mixture Y is given by

$$\sum_{j=1}^{m}\frac{p_j}{\sqrt{2\pi}\sigma_j}\exp\Big(\frac{-(x-\mu_j)^2}{2\sigma_j^2}\Big)$$

Now suppose that we are given a vector $(y_1,\ldots,y_n)$ of n realizations of the random variable Y. We wish to use the information in the y_i (and any other auxiliary information which may be at our disposal) to estimate the parameters p_j, μ_j, and σ_j.

In general there are many kind of auxiliary information which may be present in such mixture problems. For example, we may or may not know the number of groups present in the mixture. We may know some or all of the means, μ_j, or we may have some knowledge of the approximate value of some or all of the μ_j. We may know the value of some or all of the σ_j, or we may know that all the σ_j have the same value. A general routine for estimating the parameters in mixture problems should be able, as much as possible, to incorporate auxiliary information about the parameters into the estimation process.

The maximum likelihood estimates for the parameters are found by finding the correct local maximum of the log-likelihood function

$$\max_{p,\mu,\sigma}\Big\{\ \sum_{i=1}^{n}\ \log\Big[\ \sum_{j=1}^{m}\frac{p_j}{\sqrt{2\pi}\sigma_j}\exp\Big(\frac{-(y_i-\mu_j)^2}{2\sigma_j^2}\Big)\Big]\ \Big\} \tag{14.1}$$

subject to any constraints on the parameters which may be appropriate for the problem.

For mixtures of normal distributions the correct solution never occurs at the global maximum of the log-likelihood function when there is more than one group in the mixture. This is due to the fact that it is always possible to make the log-likelihood function as large as desired by setting the mean of one of the groups equal to the value of an observation and letting the corresponding standard deviation parameter tend to 0. The value of the probability density function for this group will then tend to infinity at the point in question. It follows that to find the correct solution the values of the standard deviations should be bounded away from 0.

The following code calculates the likelihood function corresponding to equation (14.1). Notice that none of the constraints which we intend to place on permissible parameter values appears in the code. The constraint conditions will be implemented elsewhere. Avoiding them at this stage produces a clean piece of code which can easily be seen to correspond to equation (14.1). The constant terms $1/\sqrt{2\pi}$ have been ignored.

```
// file: like.cpp
#include <fvar.hpp>
#include "mixture.h"

dvariable log_likelihood(dvector y,dvar_vector p,dvar_vector& mu,
dvar_vector& sd)
{
  dvariable like_fun=0.;
  int nobs=y.size();
  int ngroups=p.size();
  dvar_vector v=elem_prod(sd,sd); // Calculate the variances
  dvariable sum;
  dvariable diff;
  for (int i=1;i<=nobs;i++)
  {
    sum=0.;
    for (int j=1;j<=ngroups;j++)
    {
      diff=y(i)-mu(j);
      sum+=p(j)/sd(j)*exp(-diff*diff/(2.*v(j)));
    }
    like_fun+=log(sum+1.e-20);  // Likelihood contribution for i'th observation
  }
  return(like_fun);
}
```

The term `1.e-20` is put in to improve the numerical stability of the program. The routine `log_likelihood` is intended to be used with a function minimizing routine to estimate the parameters. In the course of such a process the function minimizer invariably will pick some parameters which fit certain observations extremely badly. An extremely bad fit is characterized by the value of `diff` being large for all groups. As a result, due to floating point underflow, `exp(-diff*diff/(2.*v(j)))` can be equal to 0 for all groups so that `sum` has the value 0. Taking the logarithm of 0 would cause a floating point exception or error. Adding the small value `1.e-20` allows the program to carry on gracefully.

It should be noted that Hasselblad grouped the observed data into size classes and analyzed the grouped data. This can easily be done and should be done if the number of observations is large, because the analysis of the grouped data can be much faster.

The function minimizer routine deals with a vector `x` of numbers. The likelihood calculation routine `log_likelihood` deals with vectors of proportions, means, and standard deviations. We need a routine to "put" the x's into the model.

```
//file: reset.cpp
#include <fvar.hpp>
#include "mixture.h"

dvariable reset(ivector& control,dvar_vector& x,dvar_vector& p, dvar_vector& mu,
    dvar_vector& sd,dvector& mumin, dvector& mumax, dvector& sdmin,
    dvector& sdmax)
```

```
{
  dvariable zpen=0.;
  int ngroups=p.size();
  int ii=1;
  if (control(1)>0)  // control(1) determines whether the p's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      p(j)=boundp(x(ii++),0.,1.,zpen);
    }
  }
  if (control(2)>0)  // control(2) determines whether the mu's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      mu(j)=boundp(x(ii++),mumin(j),mumax(j),zpen);
    }
  }
  if (control(3)>0)  // control(3) determines whether the sd's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      sd(j)=boundp(x(ii++),sdmin(j),sdmax(j),zpen);
    }
  }
  return(zpen); // Return zpen so it can be added to the function being
  // minimized
}
```

The function `reset` not only puts the x's into the parameters of the the function `like_fun`, it also determines which of the parameters are active (being estimated at the time) by examining the `ivector control` and puts constraints on the values the parameters through the function `boundp`. The function

```
dvariable boundp(dvariable x,double fmin,double fmax,dvariable& fpen)
```

bounds the variable `x` to the closed interval [`fmin,fmax`]. The term `fpen` is a penalty term which modfies the behaviour of the function. It must be added to the function being minimized. **Note that this means that if you want to maximize a function you must change the sign of the function before adding `fpen` to it!** The function `reset` supplies complete control over which parameters are active through the vector `control`. For example, if `control(1)` is equal to 1 while `control(2)` and `control(3)` are equal to 0, then only the proportions at age will be estimated while the means and standard deviations will be kept fixed. The `dvectors mumin` and `mumax` determine lower and upper bounds on the means while the `dvectors sdmin` and `sdmax` determine lower and upper bounds on the standard deviations.

Suppose that you want the standard deviations to be the same for all the groups. This can be accomplished easily by modifying the last part of `reset` to

```
  if (control(3)>0)  // control(3) determines whether the sd's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      sd(j)=boundp(x(ii),sdmin(j),sdmax(j),zpen); //Note that ii is not
                                                  // incremented
    }
    ii++;   // Now increment ii
  }
```

Another modification would be to enable individual mean or standard deviations to be fixed or active.

It should be clear that the vector x is a transitory object associated with the minimization routine. Without the function reset it is not even clear to which model parameter a particular component of x corresponds or what value of the parameter it produces. For this reason it does not makes sense to save the values of x during or after the analysis. It is the model parameters p, mu, and sd which should be saved. However this raises the question: Given the values of the model parameters, the values of the control vector, and the values of the bounds on the parameters, what should the initial values of the x vector be? This problem is solved by the functions boundpin and xinit.

```
// file: xinit.cpp
#include <fvar.hpp>
#include "mixture.h"

void xinit(ivector& control,independent_variables& x,dvector& p,dvector& mu,
    dvector& sd,dvector& mumin,dvector& mumax,dvector& sdmin,dvector& sdmax)
{
  int ngroups=p.size();
  int ii=1;
  if (control(1)>0)  // control(1) determines whether the p's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      x(ii++)=boundpin(p(j),0.,1.);
    }
  }
  if (control(2)>0)  // control(2) determines whether the mu's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      x(ii++)=boundpin(mu(j),mumin(j),mumax(j));
    }
  }
  if (control(3)>0)  // control(3) determines whether the sd's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      x(ii++)=boundpin(sd(j),sdmin(j),sdmax(j));
    }
  }
}
```

The control structure of `xinit` exactly parallels that of `reset` so that no matter what switch combinations or bounds are used, the correct values will be put into the components of `x`. The function

```
double boundpin(double x,double fmin,double fmax)
```

is the inverse of the function `boundp`. Notice that the control vector, `control`, determines how many active parameters there are.

The job of the function `nvarcal` is to use this information to calculate the number of active parameters.

```
// file: nvarcal.cpp
#include <fvar.hpp>
#include "mixture.h"

int nvarcal(ivector& control ,int ngroups)
{
  int ii=1;
  if (control(1)>0)  // control(1) determines whether the p's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      ii++;
    }
  }
  if (control(2)>0)  // control(2) determines whether the mu's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      ii++;
    }
  }
  if (control(3)>0)  // control(3) determines whether the sd's are active
  {
    for (int j=1;j<=ngroups;j++)
    {
      ii++;
    }
  }
  int nvar=ii-1;
  return(nvar);
}
```

The three functions `reset`, `xinit`, and `nvarcal` all share a common control structure and make it possible to coordinate the various users options in the estimation routine.

Here is the code for the main routine in the mixture analysis program. The data are assumed to be in a file named `mixture.dat`. The initial values for the parameters are assumed to be in a file named `mixture.par`.

```
// file: mixture.cpp
#include <fvar.hpp>
```

```
// header file containing all function prototypes used in this exmaple
#include "mixture.h"

#ifdef __BCPLUSPLUS__
  extern unsigned _stklen = 20000;
#endif
#ifdef __ZTC__
  long _stack = 20000;
#endif
void main()
{
  // The observations are in the file mixture.dat
  dvector y("mixture.dat"); // Read in the observations
  ifstream infile2("mixture.par"); // The parameter estimates are in a
   // file named mixture.par
  int ngroups;
  infile2 >> ngroups; // Read in the number of groups
  ivector control(1,10); // There are 10 control switches for this model
  infile2 >> control; // Read in the control switches
  dvector p(1,ngroups);
  dvector mu(1,ngroups);
  dvector sd(1,ngroups);
  dvector mumin(1,ngroups);
  dvector mumax(1,ngroups);
  dvector sdmin(1,ngroups);
  dvector sdmax(1,ngroups);
  infile2 >> p;  // Read in the mixture proportions
  infile2 >> mu; // Read in the mean lengths
  infile2 >> sd; // Read in the standard deviations
  infile2 >> mumin; // Read in the lower bounds on the mean lengths
  infile2 >> mumax; // Read in the upper bounds on the mean lengths
  infile2 >> sdmin; // Read in the lower bounds on the standard deviations
  infile2 >> sdmax; // Read in the upper bounds on the standard deviations
  infile2.close(); // close the file
  int nvar=nvarcal(control,ngroups); // Get the number of independent variables
  independent_variables x(1,nvar);
  double f;
  dvector g(1,nvar);
  xinit(control,x,p,mu,sd,mumin,mumax,sdmin,sdmax); // Get the initial x values
  fmm fmc(nvar);
  BEGIN_MINIMIZATION(nvar,f,x,g,fmc) // Estimate the parameters by minimizing fcomp
    f=fcomp(y,p,mu,sd,mumin,mumax,sdmin,sdmax,control,x,0);
  END_MINIMIZATION(nvar,g)
  fcomp(y,p,mu,sd,mumin,mumax,sdmin,sdmax,control,x,1); // Save the parameter
// estimates
}
```

The job of the function fcomp is to put the x values into the model by calling the function `reset` and then to evaluate the log-likelihood by calling the function `log_likelihood`.

```
// file fcomp_m.cpp
#include <fvar.hpp>
#include "mixture.h"

double fcomp(dvector y,dvar_vector p,dvar_vector& mu,dvar_vector& sd,
  dvector& mumin, dvector& mumax, dvector& sdmin, dvector& sdmax,
  ivector& control,dvar_vector& x,int print_switch)
```

```
{
  dvariable zpen;
  zpen=reset(control,x,p,mu,sd,mumin,mumax,sdmin,sdmax); // Put the x vector
                               // into the model parameters and return the
                               //bounding function penalty in zpen
  zpen+=normalize_p(p); // Make the proportions sum to 1
  if (print_switch == 1)
  {
    save_pars(p,mu,sd,mumin,mumax,sdmin,sdmax,control);
  }
  dvariable f;
  f=-log_likelihood(y,p,mu,sd); // Change the sign to minimize
  f=f+zpen; // Add the penalty from the ‘‘bounding’’ functions
  return(value(f));
}
```

Notice that while the parameterization of the mixture proportions, p, has restricted the proportions to lie between 0 and 1, the proportions have not been restricted so that they sum to 1. The function normalize_p restricts the p so that they sum to 1.

```
//file: normaliz.cpp
#include <fvar.hpp>
#include "mixture.h"

dvariable normalize_p(dvar_vector& p)
{
  dvariable psum=sum(p);
  p=p/psum;  // Now the p’s will sum to 1
  dvariable zpen=1000.*log(psum)*log(psum);
  return(zpen);
}
```

The penalty 1000.*log(psum)*log(psum) has been added to the function being minimized to remove the degeneracy in the parameterization of the p. Without this penalty the minimizing value would be independent of the value of psum. Including the penalty ensures that the minimum value will occur for the value psum= 1.

AUTODIF uses the same memory locations for the derivative calculations as are used for holding the variable objects. In consequence, after the derivatives have been calculated the current value of the parameter estimates has been lost. In order to save the results after the minimization has been carried out it is necessary to calculate the parameters one more time and print them into a file. The simplest way to do this is to use the code for fcomp which already exists, and to put a switch into fcomp so that saving the parameters is enabled. The parameters are written into the file mixture.par by the routine save_pars.

```
//file: savepar.cpp
#include <fvar.hpp>
#include "mixture.h"
```

```
void save_pars(dvar_vector& p,dvar_vector& mu,dvar_vector& sd,
  dvector& mumin, dvector& mumax,dvector& sdmin, dvector& sdmax,
  ivector& control)
{
  ofstream outfile("mixture.par");
  outfile << p.size() << "\n";  // The number of groups
  outfile << control << "\n";
  outfile << p << "\n\n";
  outfile << mu << "\n\n";
  outfile << sd << "\n\n";
  outfile << mumin << "\n\n";
  outfile << mumax << "\n\n";
  outfile << sdmin << "\n\n";
  outfile << sdmax;
}
```

Chapter 15

A Neural Network

This example illustrates the use of AUTODIF 's two and three dimensional ragged arrays.

An n-layer feed-forward neural network can be viewed as a process which, given a set of m_1 inputs $(x_{11},\ldots,x_{1m_1})$ at layer 1, produces a set of m_n outputs $(x_{n1},\ldots,x_{nm_n})$ at layer n. Level 1 is referred to as the input layer. Level n is referred to as the output layer. Levels 2 through $n-1$ are referred to as hidden layers. The quantities x_{ij} are referred to as nodes.

In a feed forward neural network the values of the nodes in each layer are determined by the values in the previous layer via the relationship

$$x_{i+1,j} = \Phi\Big(\sum_{k=1}^{m_{i-1}} w_{ijk}x_{ik} + b_{i+1,j}\Big) \tag{15.1}$$

The function Φ is referred to as a squashing function. To the best of our knowledge, the particular form of Φ does not seem to be especially important so long as it is a continuously differentiable monotone increasing function which maps $(-\infty,\infty)$ into some desired bounded set (a,b). The bounded set is often of the form $(-0.5,0.5)$ or $(0,1)$. The w_{ijk} are referred to as weights, while the b_{ij} are called bias terms.

The shape or topology of the neural net is completely determined by the number of levels, n, and the number of nodes in each level, m_i. Assume that the m_i are contained in an `ivector` object `num_nodes`.

The x_{ij} can be viewed as a two dimensional ragged array with n rows and m_i elements in the i 'th row. Such an array can be created by the declaration

```
dmatrix x(1,num_levels,1,num_nodes); // these are the nodes
```

The bias terms b_{ij} can also be viewed as a ragged matrix B with $n-1$ rows. The declaration for B requires a slight modification of the ivector num_nodes to reflect the fact that B does not need to contain row 1. The legal row index values for B should be from 2 to n.

```
ivector iv(2,num_levels);
for (int i=2;i<=num_levels;i++)
{
  iv(i)=num_nodes(i); // Put the desired elements of num_nodes into iv
}
// Now declare B
dmatrix B(2,num_levels,1,iv);
```

For each i, $1 \le i < n$, the weights, w_{ijk}, can be viewed as an m_{i+1} by m_i matrix. Thus the w_{ijk} can be viewed as a ragged three dimensional array W. To create W we need some ivector objects to contain the variable number of rows and columns.

```
ivector nrows(1,num_levels-1);
ivector ncols(1,num_levels-1);
for (i=1;i<num_levels;i++)
{
  nrows(i)=num_nodes(i+1);
  ncols(i)=num_nodes(i);
}
// Now declare W
d3_array W(1,num_levels-1,1,nrows,1,ncols);
```

If the squashing function Φ is extended to vector objects by acting elementwise on the elements of the vector, the code segment to carry out the calculations in equation (15.1) can be written as

```
for (int i=1;i<num_levels; i++)
{
  x[i+1]=Phi(W[i]*x[i]+B[i+1]);
}
```

To set up the initial vector of parameter estimates needed by the function minimizer it is necessary to copy all the values of the active parameters. To copy values into or from two and three dimensional objects one must use a series of nested loops. This process is tedious and prone to error. A collection of functions has been included in AUTODIF to carry out these sort of tasks. The function

```
set_value_inv(A,y,ii); // this is the ‘‘inverse’’ of the set_value
                       // function
```

will copy the elements of A into the dvector y starting at the offset specified by the int ii. ii will be incremented so that it points at the correct offset in y to add the values of other objects. A may be a dvector, a dmatrix, or a d3_array. For a dvector v the statements

```
int ii=6;
set_value_inv(v,y,ii);
```

will have the same effect as

```
int ii=6;
for (int i=v.indexmin();i<=v.indexmax();i++)
{
  y(ii++)=v(i);
}
```

The function

```
set_value(A,y,ii);
```

takes the values from the `dvar_vector y` beginning at the offset `ii` and puts them into the object `A`. `A` may be a `dvar_vector`, a `dvar_dmatrix`, or a `dvar3_array`. Of course it is the users responsibility to declare the vector objects `y` to have the appropriate size. To count up the number of elements in the container objects the function `size_count()` has been defined. The functions `set_value()`, `set_value_inv()`, and `size_count()` have been used in the neural net code to simplify the construction of the functions `nvar_calc` which calculates the number of active parameters to be estimated, `yinit` which puts the intial value of all the active parameters into the vector `y` for the function minimizer, and `reset` which calculates the current estimate of the active parameters from the vector `y`. The code for these functions follows.

```
// count up the number of active parameters
int nvar_calc(dmatrix& B, d3_array& W)
{
  int nvar=0;
  nvar+=size_count(B);
  nvar+=size_count(W);
  return nvar;
}

// Put the parameter values from W and B into the vector y
void yinit(d3_array& W,dmatrix& B,independent_variables& y,int& num_levels,
  ivector& num_nodes)
{
  int ii=1;
  set_value_inv(W,y,ii);
  set_value_inv(B,y,ii);
}

// Put the values from the dvar_vector y into W and B
void reset(dvar3_array W,dvar_matrix& B,dvar_vector& y,int& num_levels,
  ivector& num_nodes)
{
  int ii=1;
  set_value(W,y,ii);
  set_value(B,y,ii);
}
```

A training set for a neural net is a collection of input layer vectors together with a corresponding set of output layer vectors. For each input layer vector the corresponding output layer vector can be thought of as the desired correct output response of the network for the given input stimulus. The weights and bias terms of the network are adjusted so that the difference between desired response and the actual response is minimized. We shall assume that the training set is contained in a file called `learning.smp`. The training set will be read in every time the response of the neural net is calculated. This provides for very large training sets which could not be kept in memory.

```
ifstream infile("learning.smp");  // Open the file with the training data
infile >> num_learn;  // read in the number of examples to be used
// Loop through the training data
for (int k=1;k<= num_learn; k++)
{
  infile >> x[1];
  if (!infile)
  {
    cerr << "Error reading x[1] from file learning.smp\n";
    exit(1);
  }
  infile >> correct_response;
  if (!infile)
  {
    cerr << "Error reading correct_response from file learning.smp\n";
    exit(1);
  }
  for (int i=1;i<num_levels; i++)
  {
    x[i+1]=Phi(W[i]*x[i]+B[i+1]);
  }
  dvar_vector vdiff=correct_response-x[num_levels];
  z+=vdiff*vdiff; // z sums up the squared difference between the
                  // desired response and the actual response

  // Add the penalty terms
  dvariable v;
  v=norm(W);
  z+=.01*v*v;
  v=norm(B);
  z+=.01*v*v;
}
```

The parameters for `W` and `B` are chosen so that the total squared error is minimized. A small quadratic penalty function of the model parameters is added to keep the values of the parameters close to 0. This is much like the use of ridge regression (Hoerl and Kennard, 1970) in linear models. We don't know whether the use of such penalty functions really improves the performance of the neural net, but it seemed to produce good results for the problem considered here.

For the squashing function, Φ, we have employed the inverse tangent function `atan`.

```
dvar_vector Phi(dvar_vector& v)
{
  // This squashing function maps into the interval (-.5,.5)
  RETURN_ARRAYS_INCREMENT;  // Need this because the function returns a
                            // variable type
  int min=v.minindex();
  int max=v.maxindex();
  dvar_vector tmp(min,max)
  for (int i=min;i<=max;i++)
  {
    tmp(i)=0.31831*atan(v(i)); //0.31831 is 1/PI
    // tmp(i)=0.31831*atan(v(i))+0.5; // This maps into (0,1)
  }
  RETURN_ARRAYS_DECREMENT;
  return(tmp);
}
```

The structure of the neural net is determined by specifying the number of layers and the number of nodes in each layer. This information is put into the file `struct.nrl`.

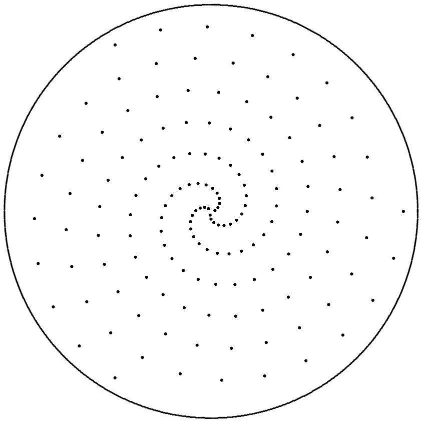
To begin training the neural net the `d3_array W` and the `dmatrix B` were filled with normally distributed random numbers, with mean 0 and standard deviation 1.0. The value of 1.0 for the standard deviation was picked by trial and error. If the standard deviation is too small the minimization routine doesn't seem to be able to find a good minimum and the procedure "stalls" at a bad local minimum.

The program `WINIT.EXE` reads the file `struct.nrl` and writes the initial parameter values into the file `WEIGHTS.PAR`.

Touretsky and Pomerlau (1989) discussed the problem of training a neural network to recognize points which lie on or near a double spiral. We thought it would be interesting to extend this problem to points which lie on or near a triple spiral.

The training set for the neural net is shown in Figure 15.1. The net is being trained to determine whether a point in a disk lies on or near one of three spirals. The input level to the net consists of two nodes, the cartesian coordinates of the point. The output level of the net has three nodes. The node corresponding to the spiral on which a particular point is located is intended to be turned on (have value +.5). and to be turned off (have value -.5) otherwise. Actually, any value for a node that is less than 0 is considered "off" while any value which is greater than 0 is considered "on". Due to the nature of the inverse tangent squashing function the maximum value produced by the neural net is about 0.42 while the minimum value is about -0.42. If the neural net has 0, 2, or 3 output nodes turned on (> 0) for a point then that point is considered unclassified.

Figure 15.1



A typical input consist the pair of coordinates of a point on the spiral $(0.0618, 0.1901)$ together with the triple $(0.5, -0.5, -0.5)$ which is the desired output for this point since it lies on the first spiral arm.

We trained neural nets of increasing complexity until a design was obtained which had enough parameters to classify all the points in the learning sets correctly. The neural net we chose has 6 layers with 2 input nodes, 8 nodes in each of the 4 hidden layers, and 3 output nodes. This neural net has 267 parameters to be estimated.

Training the neural net required about 20 hours on a 20 MHz 386 computer. The file `GRADFIL1.TMP` which stores the temporary information needed to calculate the gradient was about 3 MB in size. It was stored on a RAM disk. At any time it is possible to stop the program by typing "q". The current parameter estimates are printed into the file `EST.PAR`. To restart the learning process you must first copy the file `EST.PAR` into the file `WEIGHTS.PAR`. In this manner it is possible to carry out training sessions during periods when the computer is not required.

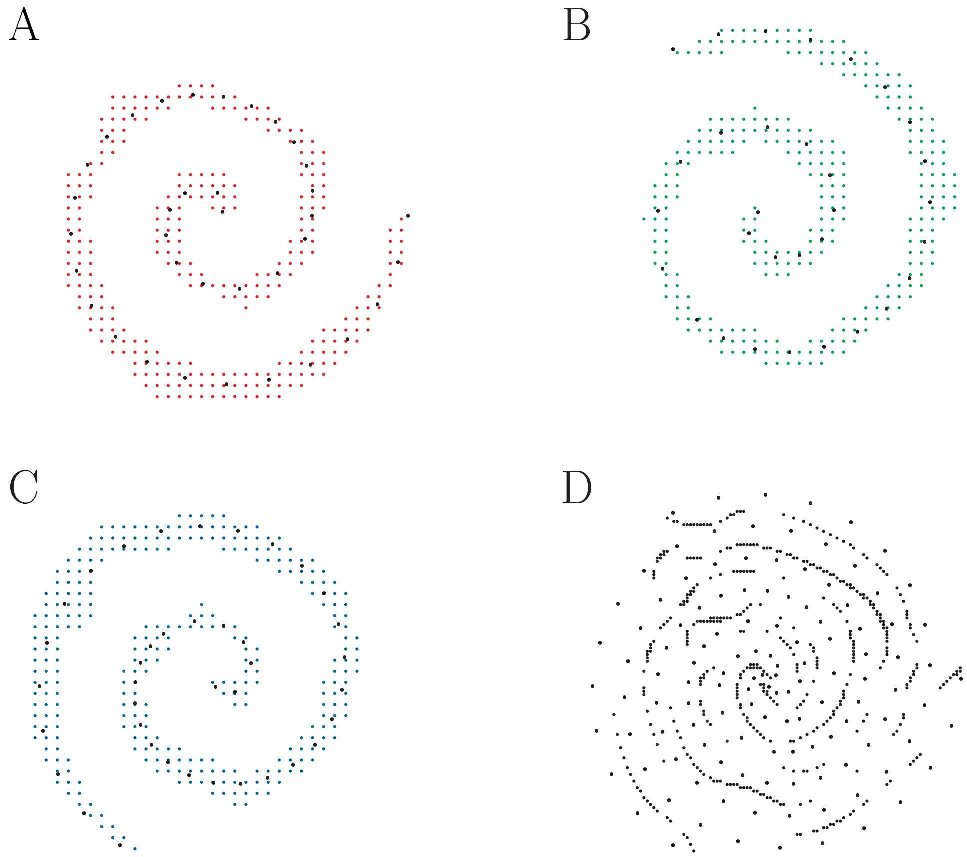


Figure 15.2

Figure 15.2 illustrates how the neural net has classified the points in the disk. The points in A, B, and C have been classified as being associated with the corresponding spiral. The small points in D are unclassified. The large points in D are the training set. The unclassified points appear to mark fairly regular boundaries between the spirals. Our solution does not seem to exhibit the “bumps and gaps” which were reported by Touretsky and Pomerlau for the double spiral solution. Perhaps this is due to the superiority of our minimization procedure over the method they employed. It may also be due to the use of more parameters combined with a small quadratic penalty function on the parameters to introduce stability into the model. In any event we seem to have obtained a nice solution to the problem.

References

Hoerl, A. E. and R. W. Kennard, 1970. Ridge regression: biased estimation for non-orthogonal problems and applications. Technometrics, 12: 55-69.

Touretzky, David S. and Dean A. Pomerlau, 1989. What's in the hidden layers? Byte 14(8): 227-233

Griewank, A. and G. F. Corliss (*eds.*), 1991. Automatic differentiation of algorithms: theory, implementation and application. Philadelphia, PA: SIAM.

Hasselblad, V. 1966. Estimation of Parameters for a mixture of normal distributions. Technometrics 8: 431-444.

Chapter 16

Index