A Programming Project for the Application of Newton's Method

Programmed by two sections of the course Programming with C++
January 2005

Abstract. Newton's method is implemented in a code called `newton`. The program reads a starting vector from a data file. The function whose zero-point is sought is defined in a subroutine as is its Jacobian in a separate subroutine. The arrays storing the starting vector, the function and its Jacobian are defined globally and dynamically. Since the dimension of these arrays is also a global variable, the code can readily be adapted to other functions. The starting vector is overwritten with an update in the Newton iteration, and thus this vector is used also for the final solution to the zero-point problem. The Newton iteration is terminated when a relative error criterion is met or when the number of iterations exceeds a limit. The relative error theshold and the maximum number of iterations are read from a data file as input. Once the solution is found within the specified accuracy, the result is written to a data file, and the number of iterations required to find it is written to a separate data file.

1 Introduction

Newton's method is a means of solving equations of the form,

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$$

where $\boldsymbol{F}$ is a given and sufficiently smooth function from $\boldsymbol{R}^n$ into $\boldsymbol{R}^n$ where $n \geq 1$. The details of the method are discussed in detail in [1] and [2], but the basic idea of Newton's method can perhaps best be illustrated by the following simple example.

Let $n = 1$ and suppose $\boldsymbol{F}(\boldsymbol{x}) \to \phi(x) = x^2$, so the solution to $\phi(x) = 0$ is $x = 0$. Suppose that the solution is initially guessed to be $x_0 = 1$. Then an improved approximation x_1 to the solution to $\phi(x) = 0$ can be obtained from an approximation of the function $\phi(x)$ which is linear in the neighborhood of x_0:

$$\phi(x_1) \approx \phi(x_0) + \phi'(x_0)(x_1 - x_0) = 0 \qquad \Rightarrow \qquad x_1 = x_0 - \phi(x_0)/\phi'(x_0).$$

The linear approximation is represented by the line $y(x) = \phi(x_0) + \phi'(x_0)(x_1 - x_0)$ which is illustrated by the cyan line in Fig. 1. The x-intercept of this line is x_1 and it represents the solution to $y(x) = 0$. Similarly an improved approximation x_2 to the solution to $\phi(x) = 0$ can be obtained from an approximation of the function ϕ which is linear in the neighborhood of x_1, and x_2 is given by the x-intercept of the green line shown in Fig. 1. The x-intercept of the red line in Fig. 1 is x_3 and so on.

Newton's method is defined in general as follows:

$$\boldsymbol{DF}(\boldsymbol{x}_{i-1})\Delta\boldsymbol{x}_i = -\boldsymbol{F}(\boldsymbol{x}_{i-1}), \qquad \boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \Delta\boldsymbol{x}_i \qquad i = 1, 2, 3, \ldots$$

where $\boldsymbol{DF}$ is the Jacobian matrix of the function $\boldsymbol{F}$, and $\Delta\boldsymbol{x}_k$ is the solution to the linear system and satisfies $\Delta\boldsymbol{x}_i = [\boldsymbol{x}_i - \boldsymbol{x}_{i-1}]$.

A particularly useful application of Newton's method is to find the extrema of a scalar-valued function. For instance, define

$$f(x, y) = \frac{x}{1 + x^2} \cdot \frac{1}{1 + y^2}$$

whose graph is shown in Fig. 2. The extrema of f are achieved at the critical points where the gradient of f vanishes:

$$\nabla f(x, y) = \left[\begin{array}{c} f_x(x, y) \\ f_y(x, y) \end{array}\right] = \left[\begin{array}{c} \dfrac{1 - x^2}{(1 + x^2)^2} \cdot \dfrac{1}{1 + y^2} \\ \dfrac{x}{1 + x^2} \cdot \dfrac{-2y}{(1 + y^2)^2} \end{array}\right].$$

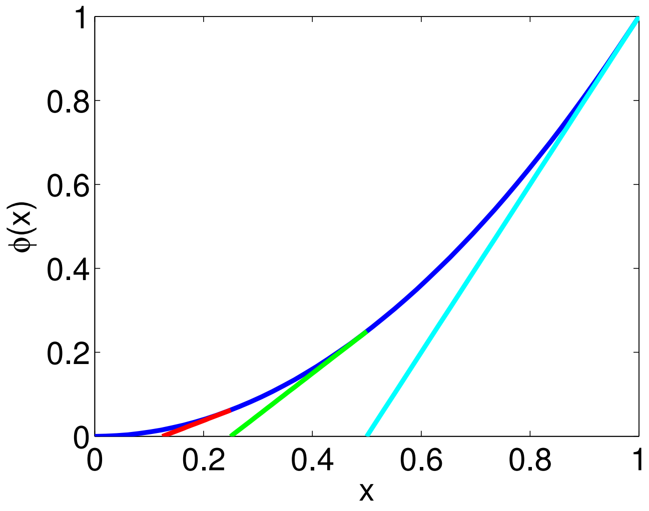

Figure 1: Graphical demonstration of three iterations of Newton's method for the function $\phi(x) = x$.

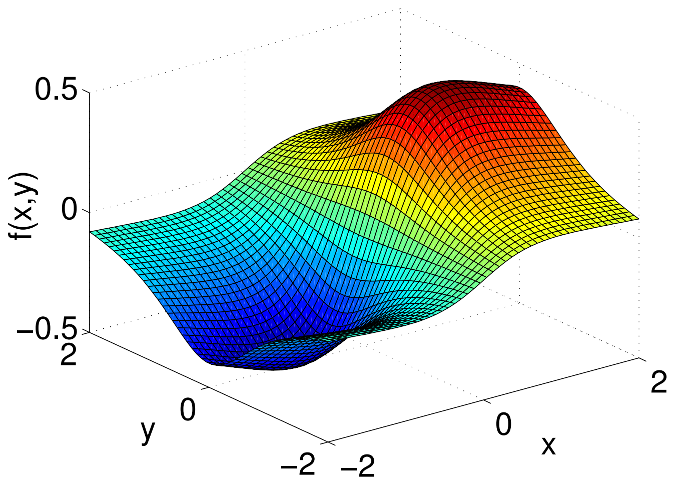

Figure 2: Plot of the scalar-valued function $f(x, y) = x/[(1 + x^2)(1 + y^2)]$.

Here f_x and f_y denote the partial derivatives of f with respect to x and y respectively. Note that f has a minimum at the critical point $(x, y) = (-1, 0)$ and a maximum at the critical point $(x, y) = (1, 0)$.

Newton's method is used to find these critical points by defining:

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{F}(x_0, x_1) = \begin{bmatrix} F_0(x_0, x_1) \\ F_1(x_0, x_1) \end{bmatrix} = \begin{bmatrix} \dfrac{1 - x_0^2}{(1 + x_0^2)^2} \cdot \dfrac{1}{1 + x_1^2} \\ \dfrac{x_0}{1 + x_0^2} \cdot \dfrac{-2x_1}{(1 + x_1^2)^2} \end{bmatrix}.$$

whose Jacobian is given by:

$$\begin{aligned} \boldsymbol{DF}(x_0, x_1) &= \begin{bmatrix} [F_0(x_0, x_1)]_{x_0} & [F_0(x_0, x_1)]_{x_1} \\ [F_1(x_0, x_1)]_{x_0} & [F_1(x_0, x_1)]_{x_1} \end{bmatrix} \\ &= \begin{bmatrix} \dfrac{2x_0(x_0^2 - 3)}{(1 + x_0^2)^3} \cdot \dfrac{1}{1 + x_1^2} & \dfrac{1 - x_0^2}{(1 + x_0^2)^2} \cdot \dfrac{-2x_1}{(1 + x_1^2)^2} \\ \dfrac{1 - x_0^2}{(1 + x_0^2)^2} \cdot \dfrac{-2x_1}{(1 + x_1^2)^2} & \dfrac{x_0}{1 + x_0^2} \cdot \dfrac{2(3x_1^2 - 1)}{(1 + x_1^2)^3} \end{bmatrix}. \end{aligned}$$

Newton's method is applied to this particular function $\boldsymbol{F}(\boldsymbol{x})$ in the present implementation of the code `newton`. Specifically, the following computed iterates,

$$\begin{array}{lcll}
\boldsymbol{x}_0 & = & \langle 0.500000, & 0.1000000\rangle \\
\boldsymbol{x}_1 & = & \langle 0.850898, & -0.0479679\rangle \\
\boldsymbol{x}_2 & = & \langle 0.974319, & 0.0015686\rangle \\
\boldsymbol{x}_3 & = & \langle 0.999052, & -1.05115\times 10^{-6}\rangle \\
\boldsymbol{x}_4 & = & \langle 0.999999, & +9.44773\times 10^{-13}\rangle \\
\boldsymbol{x}_5 & = & \langle 1.000000, & -1.71192\times 10^{-24}\rangle \\
\boldsymbol{x}_6 & = & \langle 1.000000, & 0.0000000\rangle
\end{array}$$

are ploted in Fig. 3. Note that from $\boldsymbol{x}_0 = \langle 0.5, 0.1\rangle$, 6 iterations are required to reach the critical

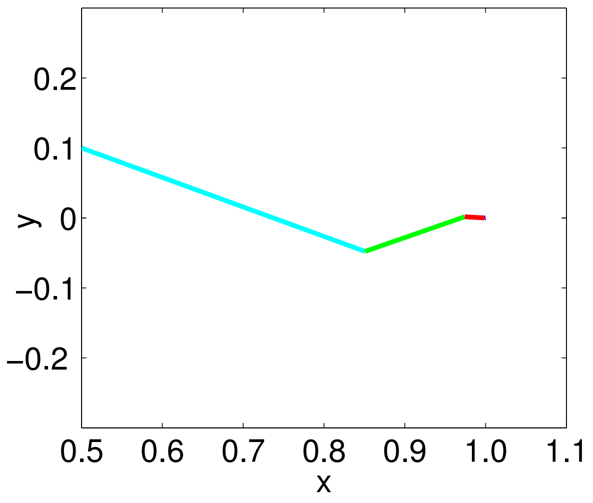

Figure 3: Plot of the Newton iterates for the scalar-valued function $f(x,y) = x/[(1+x^2)(1+y^2)]$ with initial vector $\boldsymbol{x}_0 = \langle 0.5, 0.1\rangle$.

point $\langle 1.0, 0.0\rangle$ where f has a maximum.

2 The Code from a User's Perspective

In this section it is explained how to use the code `newton` without necessarily understanding the detailed programming. Specifically, it is explained how to choose the inputs, how to interpret the outputs, and how to make the most basic changes in the code as desired.

There are two input files: `data.in` and `newton.in`. Both are text files and thus they may be generated with a usual text editor or by any code which produces the required ASCII text format.

The file `data.in` contains an initial guess vector which is the starting vector for Newton's method. The file has the following format:

```
x(0)
x(1)
...
x(n-1)
```

where `x` has the type `double*` (point to `double`) and `n` has the type `unsigned long`.

The file `newton.in` contains the following input parameters in the following format:

```
tol
imax
```

where `tol` has the type `double` and `imax` has the type `unsigned long`. The first parameter `tol` is the relative error tolerance which defines the following stopping criterion for the Newton iteration:

$$\frac{\|\Delta \boldsymbol{x}_{i-1}\|}{\|\boldsymbol{x}_i\|} \leq \texttt{tol}.$$

Since $\boldsymbol{x}$ and $\Delta\boldsymbol{x}$ have type `double*`, `tol` should not be selected smaller than double precision accuracy. The second parameter `imax` specifies the maximum number of iterations, so the Newton iteration index i never exceeds `imax`. Thus, `imax` defines the following stopping criterion for the Newton iteration:

$$i > \texttt{imax}.$$

When either of the last two inequalities is met, the Newton iteration terminates. The total number of iterations performed is stored in the output variable `maxi` discussed below.

There are two output files: `data.out` and `newton.out`. Both are text files and thus they may be read with a usual text editor or by any code which interprets the specified ASCII text format.

The file `data.out` contains the final vector computed after `maxi` Newton iterations. The file has the same format as `data.in`:

```
x(0)
x(1)
...
x(n-1)
```

where `x` has the type `double*` and `n` has the type `unsigned long`. Since `data.in` and `data.out` have the same format, `data.out` may be copied to `data.in` and the code may be restarted to perform further iterations. Such a restart may in fact be necessary when the Newton iterations are stopped by the maximum allowed iterations instead of by the relative error criterion. Whether the maximum allowed iterations has been reached can be checked in the next output file discussed below. On the other hand, if the starting vector generates a divergent sequence, then restarting the code after copying `data.out` to `data.in` will not lead to convergence. Divergence should also be recognizable from very large values in `data.out`. Since Newton's method is only guaranteed to converge when started sufficiently close to a solution, divergence is a distinct possibility; see the detailed example shown below at the end of Section 4. If a good initial estimate of the solution cannot be found, then another method should be used to get into the neighborhood of a solution, and then Newton's method can take over to provide very rapid convergence.

The file `newton.out` contains `maxi` written as follows:

```
Number of iterations performed: maxi
```

where `maxi` has the type `unsigned long`. Furthermore, if `maxi = imax` holds, then the following warning is written to `newton.out`:

```
WARNING:  The number of iterations performed
has reached the maximum allowed.  Increase imax
or tol or seek a nearer starting vector.
```

To use `newton` to solve a different problem requires to change some subroutines. Note that the dimension of the problem is defined in `newton.cpp` with `n = 2`. Note further that `F` is calculated in `calcF.cpp` and `DF` is calculated in `calcDF.cpp`. Thus, a new `F` and its Jacobian `DF` must be reflected in new functions `calcF.cpp` and `calcDF.cpp`, and a new problem dimension requires a change in `n` in `newton.cpp`.

3 The Code from a Developer's Perspective

In this section, further details of the programming are described. From `newton.hpp` it can be seen that only the functions `norm` and `solve` have only local inputs or outputs. These two functions are *object oriented* in the sense that they are general enough to be used in a broader context. The other functions in this project are narrowly focused to solve the problem at hand, and these narrowly focused functions communicate through global variables while having `void` input and `void` output.

The two object oriented functions, `norm` and `solve`, can be described as follows. The function `norm` has the declaration:

```
double norm(double *y,unsigned long m)
```

and the purpose of the function is to compute the Euclidean norm of the `m`-dimensional vector `y`. Note that the norm is computed in double precision and that `math.h` is required to compute `sqrt`. The function `solve` has the declaration:

```
void solve(double **A,double *b,double *x,unsigned long n)
```

and the purpose of the function is to solve the `n`-dimensional linear system `Ax = b` for the unknown vector `x` where `A` and `b` are given. Note that the system is solved in double precision and the system size is limited by the maximum possible `unsigned long`. The solution technique implemented in `solve` is Guass Elimination with partial pivoting (row exchange) strategy. Notice in the lines of `solve`,

```
        ptm    = A[k];
        A[k]   = A[kiv];
        A[kiv] = ptm;

        btm    = b[k];
        b[k]   = b[kiv];
        b[kiv] = btm;
```

that rows can be easily exchanged with the use of pointers. Indices are defined throughout the project as `unsigned long` because they are compared with `n` which has the type `unsigned long`. On this basis, the auxillary index `ip` is defined at the end of `solve` in order to avoid a negative index in the final for-loop.

The remaining functions communicate through the following global variables:

```
double *x, *F, **DF, tol;
unsigned long imax, maxi, n;
```

which are defined as global in the function `newton.cpp`, highest in the structure chart, while they are similarly declared with `extern` in `newton.hpp`, included by most other functions. These global variables receive dynamically reserved storage space according to the following in `newton.cpp`:

```
    n = 2;
    x = new double[n];
    F = new double[n];
    DF = new double*[n];
    for (i=0;i<n;i++) DF[i] = new double[n];
```

and this storage space is also deleted at the end of `newton.cpp`:

```
    delete x;
    delete F;
    for (i=0;i<n;i++) delete DF[i];
    delete DF;
```

In the next level of the structure chart, the functions `init`, `body` and `term` are called. The two input files `data.in` and `newton.in` are read in `init` and the two output files `data.out` and `newton.out` are written in `term`. The Newton iteration is actually performed in `body`:

```
    maxi = 0;
    do
    {
        calcF();
        calcDF();
        for (i=0;i<n;i++) F[i] = -F[i];
        solve(DF,F,dx,n);
        for (i=0;i<n;i++) x[i] += dx[i];
        ndx = norm(dx,n);
        nx  = norm(x,n);
        maxi++;
    } while ((ndx > tol*nx) && ((maxi+1) <= imax));
```

Note that with a do-while loop, at least one iteration is performed since the stopping criteria are situated at the end. The linear system which must be solved in each iteration is `DF dx = -F`, and `x` is updated through `x = x + dx`. Since `F` and `DF` depend upon `x`, they are updated by calls to `calcF` and `calcDF` respectively. Notice that `-F` cannot be passed to `solve` through `solve(DF,-F,dx,n)` since `F` has the type `double*` and `-F` is technically a negative address. Therefore, the *values* of `F` must be negated,

```
        for (i=0;i<n;i++) F[i] = -F[i];
```

before the call to `solve`:

```
        solve(DF,F,dx,n);
```

Then `x` is updated according to

```
        for (i=0;i<n;i++) x[i] += dx[i];
        ndx = norm(dx,n);
        nx  = norm(x,n);
```

where the norms of `dx` and `x` are computed respectively in the last two lines and used in the do-while continuation criterion `(ndx > tol*nx)` seen below. The number `maxi` of iterations performed is incremented by `maxi++`, and to avoid that `maxi` ever exceeds `imax` the associated do-while continuation criterion is `((maxi+1) <= imax)` as seen in the following:

```
    maxi = 0;
    do
    {
        ...
        maxi++;
    } while ((ndx > tol*nx) && ((maxi+1) <= imax));
```

Note that `dx` need only be defined locally in `body` and thus it is allocated dynamically and deleted in `body`.

The makefile groups the (object-oriented) functions, `norm` and `solve`, together in `SCR1` as they do not depend upon the global variables but rather only on their local constructions. All other functions are grouped together in `SRC2` as they all depend upon the include file `INC = newton.hpp` which contains the prototypes and `extern` statements. The object files of `SCR1` are grouped together in `OBJ1` and the object files of `SCR2` are grouped together in `OBJ2`, so only `OBJ2` depends upon `INC`. The executable file `newton.exe` depends upon the `OBJ1` and `OBJ2` files. The executable is built by typing `make all`. The directory can be cleaned by typing `make clean`.

4 Sample Results

At the end of the Introduction, a sequence of iterates $\{\boldsymbol{x}_i : i = 0, \ldots, 6\}$ is listed which were computed with `newton`. The following is a list of iterates computed with `newton` but now starting with the initial vector $\boldsymbol{x}_0 = \langle 0.5, 0.5\rangle$:

$$\begin{array}{rcll}
\boldsymbol{x}_0 & = & \langle 0.50000, & +0.5000\rangle \\
\boldsymbol{x}_1 & = & \langle 6.12500, & -18.8750\rangle \\
\boldsymbol{x}_2 & = & \langle 7.81093, & -23.5422\rangle \\
\boldsymbol{x}_3 & = & \langle 9.88039, & -29.3916\rangle \\
\boldsymbol{x}_4 & = & \langle 12.4407, & -36.7128\rangle \\
\boldsymbol{x}_5 & = & \langle 15.6216, & -45.8706\rangle \\
\boldsymbol{x}_6 & = & \langle 19.5829, & -57.3225\rangle \\
\boldsymbol{x}_7 & = & \langle 24.5229, & -71.6408\rangle \\
\boldsymbol{x}_8 & = & \langle 30.6890, & -89.5412\rangle \\
\boldsymbol{x}_9 & = & \langle 38.3893, & -111.919\rangle \\
\boldsymbol{x}_{10} & = & \langle 48.0091, & -139.892\rangle \\
\ldots & & & \\
\boldsymbol{x}_{90} & = & \langle 2.71948\times 10^{+09}, & -7.91700\times 10^{+09}\rangle \\
\boldsymbol{x}_{91} & = & \langle 3.39934\times 10^{+09}, & -9.89625\times 10^{+09}\rangle \\
\boldsymbol{x}_{92} & = & \langle 4.24918\times 10^{+09}, & -1.23703\times 10^{+10}\rangle \\
\boldsymbol{x}_{93} & = & \langle 5.31148\times 10^{+09}, & -1.54629\times 10^{+10}\rangle \\
\boldsymbol{x}_{94} & = & \langle 6.63935\times 10^{+09}, & -1.93286\times 10^{+10}\rangle \\
\boldsymbol{x}_{95} & = & \langle 8.29918\times 10^{+09}, & -2.41608\times 10^{+10}\rangle \\
\boldsymbol{x}_{96} & = & \langle 1.03740\times 10^{+10}, & -3.02010\times 10^{+10}\rangle \\
\boldsymbol{x}_{97} & = & \langle 1.29675\times 10^{+10}, & -3.77512\times 10^{+10}\rangle \\
\boldsymbol{x}_{98} & = & \langle 1.62093\times 10^{+10}, & -4.71890\times 10^{+10}\rangle \\
\boldsymbol{x}_{99} & = & \langle 2.02617\times 10^{+10}, & -5.89863\times 10^{+10}\rangle \\
\boldsymbol{x}_{100} & = & \langle 2.53271\times 10^{+10}, & -7.37328\times 10^{+10}\rangle
\end{array}$$

The sequence generated with this starting vector is clearly divergent. In fact most starting vectors in $\boldsymbol{R}^2$ generate divergent sequences for the function at hand.

This point is demonstrated graphically in Fig. 4 which represents starting vectors that generate convergent or divergent Newton iterations for the present function f. The black dots represent points leading to divergent sequences, and colored asterisks represent starting vectors generating sequences which converge within one to seven iterations where the color is coded to the number iterations performed. Note that only one iteration was performed for the starting vector $\boldsymbol{x}_0 = \langle 0, 0\rangle$; however, a correct critical point for f was not found. This failure occurs because the Jacobian satisfies $\boldsymbol{DF} = \boldsymbol{0}$ at the origin $\langle 0, 0\rangle$, and after a single iteration the new iterate is $\boldsymbol{x}_1 = \langle \texttt{nan,nan}\rangle$, where `nan` stands for *not a number.* For all other starting vectors represented in Fig. 4, the Jacobian is non-singular. For the black points, the number `maxi` of iterations performed reached the specified limit `imax = 100`. For the colored asterisks, the convergence criterion was met with respect to the input `tol = 1.0e-7`, and there were no convergent iterations requiring more than seven iterations.

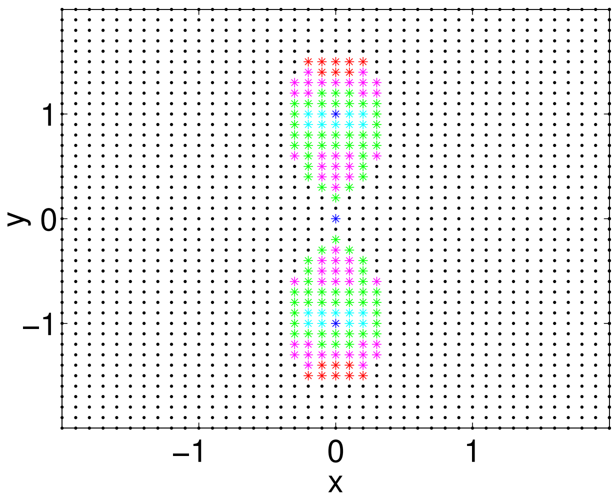

Figure 4: Graphical representation of starting vectors which generate convergent or divergent Newton iterations for the scalar-valued function $f(x,y) = x/[(1+x^2)(1+y^2)]$. Black dots represent points leading to divergent sequences. Colored asterisks represent starting vectors generating sequences which converge within: 1 (blue), 4 (cyan), 5 (green), 6 (magenta), or 7 (red) iterations.

References

[1] P. Deuflhard und A. Hohmann, *Numerische Mathematik I*, Walter de Gruyter, Berlin, 1993.

[2] G. Hämmerlin und K.-H. Hoffmann, *Numerische Mathematik*, Springer-Verlag, Berlin, 1980.

A ReadMe for newton

```
                     Summary of newton, a code to
                     ----------------------------
                       implement Newton's method
                       -------------------------

Created by two sections of the
course Programming with C++
January 2005

Abstract
--------
    Newton's method is implemented in a code called newton.  The
program reads a starting vector (x) from a data file.  The function
(F) whose zero-point is sought is defined in a subroutine as is its
Jacobian (DF) in a separate subroutine.  The arrays storing the
starting vector, the function and its Jacobian are defined globally
and dynamically.  Since the dimension (n) of these arrays is also a
global variable, the code can readily be adapted to other functions.
The starting vector is overwritten with an update in the Newton
iteration, and thus this vector is used also for the final solution to
the zero-point problem.  The Newton iteration is terminated when a
relative error criterion is met or when the number of iterations
exceeds a limit.  The relative error theshold (tol) and the maximum
number of iterations (imax) are read from a data file as input.  Once
the solution is found within the specified accuracy, the result is
written to a data file, and the number of iterations (maxi) required
to find it is written to a separate data file.

MyProgram from a User's Perspective.
-----------------------------------
    There are two input files for newton: data.in, which contains the
initial vector, x[i], i=0,(n-1), and newton.in, which contains the
input parameters tol and imax.  There are also two output files for
newton: data.out, which contains the solution vector, x[i], i=0,(n-1),
and newton.out, which contains maxi.  The format of the input and the
output can be altered in init and in term.  All data files are text
files and can be written or read with a usual text editor.

Input File:  data.in
--------------------
    The input file data.in is a text file with the following format
which can be seen in init:

          x[0]
          x[1]
          ...
          x[n-1]

where x is a pointer to double, and its elements are established
dynamically.  Note that n and x are global variables.
```

```
Input File:  newton.in
----------------------
    The input file newton.in is a text file with the following format
which can be seen in init:

          tol
          imax

where tol is double and imax is unsigned long.  Note that tol and
imax are global variables.  tol is used in the stopping criterion:
          ||dx|| <= tol * ||x||
where the Newton iteration involves the solution to the linear
system:
          DF dx = -F,     x = x + dx
imax is used in the stopping criterion:
          (maxi+1) > imax
where (maxi+1) would be the number of the next iteration.

Output File:  data.out
----------------------
    The output file data.out is a text file with the following format
which can be seen in term:

          x[0]
          x[1]
          ...
          x[n-1]

where x is a pointer to double, and its elements are established
dynamically.  Note that n and x are global variables.  Since data.out
and data.in have the same format, data.out can be copied to data.in
to restart the code and to perform further iterations.  If the values
in data.out are excessively large it is likely that the Newton iteration
with the specified initial vector is divergent.  In this case another
starting vector should be selected.

Output File:  newton.out
----------------------
    The output file newton.out is a text file with the following format
which can be seen in term:

          Number of iterations performed: maxi

where maxi is unsigned long.  Note that maxi is a global variable.
If maxi = imax holds, then the following warning is written to
newton.out:
          WARNING:  The number of iterations performed
          has reached the maximum allowed.  Increase imax
          or tol or seek a nearer starting vector.

Standard Output:
```

Only error messages are reported on the screen, and these may occur when files are not correctly handled.

Compilation

The makefile groups the (object-oriented) functions together in SCR1 which do not depend upon the global variables but rather only on their local constructions. All other functions are grouped together in SRC2 as they all depend upon the include file INC = newton.hpp which contains the prototypes and extern statements. Note that all global variables are declared, and established dynamically as necessary, in newton.cpp. The object files of SCR1 are grouped together in OBJ1 and the object files of SCR2 are grouped together in OBJ2, so only OBJ2 depends upon INC. The executable file newton.exe depends upon the OBJ1 and OBJ2 files. The executable is built by typing "make all". The directory can be cleaned by typing "make clean".

B Structure Chart for newton

```
                   StructoGram for newton

                        +---------+
                        | newton  |
                        +---------+
                         /   |   \
                        /    |    \
                +------+  +------+ +------+
                | init |  | body | | term |
                +------+  +------+ +------+
                             |
                             |
               +---------+---+----+-------+
               |         |        |       |
           +------+  +------+ +------+ +------+
           | calcF|  |calcDF| | solve| | norm |
           +------+  +------+ +------+ +------+
```

C Data Flow for newton

```
                        Data Flow for newton

             LocalIn  LocalOut  GlobalIn      GlobalOut  Read        Written/Printed
body:                           x,n,tol,imax  x, maxi
calcDF:                         x             DF
calcF:                          x             F
init:                                                    x,imax,tol
newton:                         n,x,F,DF
norm:        y,m      sum
solve:       A,b,n    x
term:                           n,x                                  x,maxi
```

D Data Dictionary for `newton`

Data Dictionary for newton

`A:`	`Matrix, local in solve`
`DF:`	`Jacobian of F, global and dynamic`
`F:`	`Function whose zero-point is sought, global and dynamic`
`b:`	`Right-hand side, local in solve`
`body:`	`Function with Newton iteration`
`btm:`	`Variable to swap b-components, local in solve`
`calcDF:`	`Function to calculate the Jacobian DF`
`calcF:`	`Function to calculate the function F`
`data.in:`	`Input file with start vector for Newton iteration`
`data.out:`	`Input file with solution vector from Newton iteration`
`dx:`	`Solution vector in Newton iteration, local in body`
`i:`	`Loop variable used locally everywhere`
`iFILE1:`	`Input file class used locally in init to read data.in`
`iFILE2:`	`Input file class used locally in init to read newton.in`
`imax:`	`Upper limit on Newton iterations`
`init:`	`Function in which input is read`
`ip:`	`Loop variable used locally in solve`
`j:`	`Loop variable used locally everywhere`
`k:`	`Loop variable used locally in solve`
`kiv:`	`Pivot index, local in solve`
`m:`	`Dimension of vector, local in norm`
`maxi:`	`Total number of Newton iterations used`
`n:`	`Dimension of x and F`
`ndx:`	`Norm of dx, local in norm`
`newton:`	`Main program`
`norm:`	`Function to calculate norm of input`
`nx:`	`Norm of x, local in norm`
`oFILE1:`	`Output file class used locally in term to write data.out`
`oFILE2:`	`Output file class used locally in term to write newton.out`
`piv:`	`Pivot value, local in solve`
`ptm:`	`Variable to swap A-rows, local in solve`
`solve:`	`Funtion for solving a linear system with Guass Elimination`
`sum:`	`Used for vector norm, local in norm`
`term:`	`Function in which output is written`
`tol:`	`Relative error stopping threshold for Newton iteration`
`x:`	`Vector argument of F, global and dynamic`
`y:`	`Input vector, local in norm`

E makefile for newton

```
# Command macros:
CPP = bcc32
OPTS =
DEL = del

# File group macros:
INC = \
        newton.hpp

SRC1 = \
        norm.cpp \
        solve.cpp

SRC2 = \
        body.cpp \
        calcDF.cpp \
        calcF.cpp \
        init.cpp \
        newton.cpp \
        term.cpp

OBJ1 = $(SRC1:.cpp=.obj)
OBJ2 = $(SRC2:.cpp=.obj)

EXE = \
        newton.exe

# Make everything up-to-date:
all: $(EXE)

# The executable file depends upon the object files and actions follow:
$(EXE): $(OBJ1) $(OBJ2)
        $(CPP) $(OPTS) -e$(EXE) $(OBJ1) $(OBJ2)

# Implicit rule for building object files from source files:
.cpp.obj:
        $(CPP) $(OPTS) -c $*.cpp

# Object files depend upon include files:
$(OBJ2): $(INC)

# Clean up the directory:
clean:
        -@if exist *.obj $(DEL) *.obj > nul
        -@if exist *.tds $(DEL) *.tds > nul
        -@if exist *.exe $(DEL) *.exe > nul
```