Introduction to numerical projects

Here follows a brief recipe and recommendation on how to write a report for each project.

- Give a short description of the nature of the problem and the eventual numerical methods you have used.
- Describe the algorithm you have used and/or developed. Here you may find it convenient to use pseudocoding. In many cases you can describe the algorithm in the program itself.
- Include the source code of your program. Comment your program properly.
- If possible, try to find analytic solutions, or known limits in order to test your program when developing the code.
- Include your results either in figure form or in a table. Remember to label your results. All tables and figures should have relevant captions and labels on the axes.
- Try to evaluate the reliabilty and numerical stability/precision of your results. If possible, include a qualitative and/or quantitative discussion of the numerical stability, eventual loss of precision etc.
- Try to give an interpretation of you results in your answers to the problems.
- Critique: if possible include your comments and reflections about the exercise, whether you felt you learnt something, ideas for improvements and other thoughts you've made when solving the exercise. We wish to keep this course at the interactive level and your comments can help us improve it.
- Try to establish a practice where you log your work at the computerlab. You may find such a logbook very handy at later stages in your work, especially when you don't properly remember what a previous test version of your program did. Here you could also record the time spent on solving the exercise, various algorithms you may have tested or other topics which you feel worthy of mentioning.

Format for electronic delivery of report and programs

The preferred format for the report is a PDF file. You can also use DOC or postscript formats. As programming language we prefer that you choose between C/C++, Fortran90/95 or Python. The following prescription should be followed when preparing the report:

- Use Fronter at `http://blyant.uio.no` to hand in your projects, log in at blyant.uio.no and choose 'fellesrom fys3150'. Thereafter you will see an icon to the left with 'hand in' or 'innlevering'. Click on that icon and go to the given project. There you can load up the files within the deadline.
- Upload **only** the report file and the source code file(s) you have developed. The report file should include all of your discussions and a list of the codes you have developed. Do not include library files which are available at the course homepage, unless you have made specific changes to them.

- Comments from us on your projects, approval or not, corrections to be made etc can be found under your Fronter domain (`http://blyant.uio.no`) and are only visible to you and the teachers of the course.

Finally, we encourage you to work two and two together. Optimal working groups consist of 2-3 students. You can then hand in a common report.

Project 1, deadline Wednesday 14 september 12am (midnight)

The aim of this project is to get familiar with various matrix operations, from dynamic memory allocation to the usage of programs in the library package of the course. For Fortran users memory handling and most matrix and vector operations are included in the ANSI standard of Fortran 90/95. Array handling in Python is also rather trivial. For C++ user however, there are three possible options

1. Make your own functions for dynamic memory allocation of a vector and a matrix. Use then the library package lib.cpp with its header file lib.hpp for obtaining LU-decomposed matrices, solve linear equations etc.

2. Use the library package lib.cpp with its header file lib.hpp which includes a function `matrix` for dynamic memory allocation. This program package includes all the other functions discussed during the lectures for solving systems of linear equations, obatining the determinant, getting the inverse etc.

3. Finally, we provide at the lab a library package which uses Blitz++'s classes for array handling. You could then, since Blitz++ is installed on all machines at the lab, use these classes for handling arrays. Alternatively, you can install Blitz++ on your laptop/PC from `http://www.oonumerics.org/blitz/`.

Your program, whether it is written in C++, Python or Fortran 90/95, should include dynamic memory handling of matrices and vectors.

The material needed for this project is covered by chapter 6 of the lecture notes, in particular section 6.4 and subsequent sections.

Many important differential equations in the Sciences can be written as linear second-order differential equations

$$\frac{d^2y}{dx^2} + k^2(x)y = f(x),$$

where f is normally called the inhomogeneous term and k^2 is a real function.

A classical equation from electromagnetism is Poisson's equation. The electrostatic potential Φ is generated by a localized charge distribution $\rho(\mathbf{r})$. In three dimensions it reads

$$\nabla^2\Phi = -4\pi\rho(\mathbf{r}).$$

With a spherically symmetric Φ and $\rho(\mathbf{r})$ the equations simplifies to a one-dimensional equation in r, namely

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\rho(r),$$

which can be rewritten via a substitution $\Phi(r) = \phi(r)/r$ as

$$\frac{d^2\phi}{dr^2} = -4\pi\rho(r).$$

The inhomogeneous term f or source term is given by the charge distribution ρ

We will rewrite this equation by letting $\phi \to u$ and $r \to x$. The general one-dimensional Poisson equation reads then

$$-u''(x) = f(x).$$

(a) In this project we will solve the one-dimensional Poissson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations.

To be more explicit we will solve the equation

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0.$$

and we define the discretized approximation to u as v_i with grid points $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$. The step length or spacing is defined as $h = 1/(n+1)$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. We approximate the second derivative of u with

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n,$$

where $f_i = f(x_i)$. Show that you can rewrite this equation as a linear set of equations of the form

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}},$$

where $\mathbf{A}$ is an $n \times n$ tridiagonal matrix which we rewrite as

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{pmatrix} \tag{1}$$

and $\tilde{b}_i = h^2 f_i$.

In our case we will assume that the source term is $f(x) = 100e^{-10x}$, and keep the same interval and boundary conditions. Then the above differential equation has an analytic solution given by $u(x) = 1-(1-e^{-10})x-e^{-10x}$ (convince yourself that this is correct by inserting the solution in the Poisson equation). We will compare our numerical solution with this analytic result in the next exercise.

(b) We can rewrite our matrix $\mathbf{A}$ in terms of one-dimensional vectors a, b, c of length $1:n$. Our linear equation reads

$$\mathbf{A} = \begin{pmatrix} b_1 & c_1 & 0 & \ldots & \ldots & \ldots \\ a_2 & b_2 & c_2 & \ldots & \ldots & \ldots \\ & a_3 & b_3 & c_3 & \ldots & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \ldots \\ \ldots \\ \ldots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \ldots \\ \ldots \\ \ldots \\ \tilde{b}_n \end{pmatrix}. \tag{2}$$

A tridiagonal matrix is a special form of banded matrix where all the elements are zero except for those on and immediately above and below the leading diagonal. The above tridiagonal system can be written as

$$a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i, \tag{3}$$

for $i = 1, 2, \ldots, n$. The algorithm for solving this set of equations is rather simple and requires two steps only, a decomposition and forward substitution and finally a backward substitution.

Your first task is to set up the algorithm for solving this set of linear equations. Find also the precise number of floating point operations needed to solve the above equations. Compare this with standard Gaussian elimination and LU decomposition.

Then you should code the above algorithm and solve the problem for matrices of the size 10×10, 100×100 and 1000×1000. That means that you choose $n = 10$, $n = 100$ and $n = 1000$ grid points.

Compare your results (make plots) with the analytic results for the different number of grid points in the interval $x \in (0, 1)$. The different number of grid points corresponds to different step lengths h.

(c) Compute the relative error in the data set $i = 1, \ldots, n$,by setting up

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

as function of $log_{10}(h)$ for the function values u_i and v_i. For each step length extract the max value of the relative error. Try to increase n to $n = 10000$ and $n = 10^5$. Make a table of the results and comment your results.

(d) Compare your results with those from the LU decomposition codes for the matrix of sizes 10×10, 100×100 and 1000×1000. Here you should use the library functions provided on the webpage of the course. Use for example the unix function *time* when you run your codes and compare the time usage between LU decomposition and your tridiagonal solver. Alternatively, you can use the functions in C++, Fortran or Python that measure the time used.

Make a table of the results and comment the differences in execution time How many floating point operations does the LU decomposition use to solve the set of linear equations? Can you run the standard LU decomposition for a matrix of the size $10^5 \times 10^5$? Comment your results.

To compute the elapsed time in c++ you can use the following statements

Time in C++

```
using namespace std;
...
#include "time.h"    //  you have to include the time.h header
int main()
{
    // declarations of variables
    ...
```

```
    clock_t start, finish;  //  declare start and final time
    start = clock();
    // your code is here, do something and then get final time
    finish = clock();
    ( (finish - start)/CLOCKS_PER_SEC );
...
```

Similarly, in Fortran, this simple example shows how to compute the elapsed time.

Time in Fortran

```
PROGRAM time
 REAL :: etime              ! Declare the type of etime()
 REAL :: elapsed(2)         ! For receiving user and system time
 REAL :: total              ! For receiving total time
 INTEGER :: i, j

 WRITE(*,*) 'Start'

 DO i = 1, 5000000
       j = j + 1
 ENDDO

 total = ETIME(elapsed)
 WRITE(*,*) 'End: total=', total, ' user=', elapsed(1), &
               ' system=', elapsed(2)

END PROGRAM time
```

(e) The last exercise is optional (frivillig). The aim here is to test possible memory strides when performing operations on matrices. To be more specific, we will look at the problem of matrix matrix multiplications. In C++ matrix elements are ordered in a row-major way while in Fortran they are ordered in a column-major order. The task here is to write a small program which sets up two random (use the ran0 function in the library lib.cpp to initialize the matrix) double precision valued matrices of dimension $10^4 \times 10^4$.

The matrix multiplication of two matrices $\mathbf{A} = \mathbf{BC}$ could then take the following form in C++

```
   for(i=0 ; i < n ; i++) {
      for(j=0 ; j < n ; j++) {
         for(k=0 ; k < n ; k++) {
            a[i][j]+=b[i][k]*c[k][j]
         }
      }
   }
```

and in Fortran we have

```
   DO  j=1,  n
      DO i=1, n
         DO k = 1, n
            a(i,j)=a(i,j)+b(i,k)*c(k,j)
```

```
        ENDDO
    ENDDO
ENDDO
```

However, Fortran has an intrisic function called MATMUL, and the above three loops can be coded in a single statement a=MATMUL(b,c).

After you have initialized the matrices **B** and **C** compute the matrix **A** using the above algorithms. Compute the time needed for the matrix multiplications (do not include the time needed for the set up of the matrices and their initializations). Then we aim at testing a possible memory stride. To do this, we interchange the order of the loop variables i and j in the C++ and Fortran codes. Do you see a change in time? For Fortran users you could also try to use the inbuilt function a=MATMUL(b,c) and see if that runs faster than your algorithm. You can also use the functions in BLAS to compute the matrix-matrix multiplications.

If you have time, you could compare your C++ or Fortran program with matrix multiplication in Python.

Good luck.