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 or as an ipython notebook file. As programming language we prefer that you choose between C/C++, Fortran2008 or Python. The following prescription should be followed when preparing the report:

- Use Devilry to hand in your projects, log in at `http://devilry.ifi.uio.no` with your normal UiO username and password. There you can load up the files within the deadline.
- Upload **only** the report file! For the source code file(s) you have developed please provide us with your link to your github domain. 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.

- In your git repository, please include a folder which contains selected results. These can be in the form of output from your code for a selected set of runs and input parameters.
- Comments from us on your projects, approval or not, corrections to be made etc can be found under your Devilry domain 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, Challenge version, deadline Monday 19 September 12pm (noon)

This project has many of the same aims as the normal version but it contains also additional challenges. In particular, those of you who are familiar with numerical algorithms behind Gaussian elmination and C++ programming could attempt at studying this alternative project. The difficulty of this project is a minor step up from the standard project, but the extra difficulty lies almost exclusively in the mathematics and the handling of differential equations and their boundary conditions. The programming involved is no more difficult than in the standard project 1.

The programming aim of this project is to get familiar with various vector and 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 several possible options. Two are listed here.

1. For this exercise we recommend that you make your own functions for dynamic memory allocation of a vector and a matrix. You don't need to write a class for this operations. Use then the library package lib.cpp with its header file lib.hpp for obtaining LU-decomposed matrices, solve linear equations etc.
2. A very good and often recommended library for C++ handling of arrays is the library Armadillo, to be found at `arma.sourceforge.net`. We will discuss the usage of this library during the lab sessions and lectures. Armadillo has also an interface to Lapack functions for solving systems of linear equations.

Your program, whether it is written in C++, Python or Fortran2008, 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

$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + k^2(x)y = f(x), \tag{1}$$

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

A classical equation from electromagnetism is PoissonâĂŹs equation. The electrostatic poten- tial Ψ is generated by a localized charge distribution $\rho(r)$. In three dimensions it reads

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

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

$$\frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}\Psi}{\mathrm{d}r}\right) = -4\pi\rho(r), \tag{3}$$

which can be rewritten via a substitution $\Psi(r) = \psi(r)/r$ as

$$\frac{\mathrm{d}^2\psi}{\mathrm{d}r^2} = -4\pi r\rho(r). \tag{4}$$

The inhomogeneous term f or source term is given by the charge distribution ρ multiplied by r and the constant 4π.

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

$$-u''(x) = f(x). \tag{5}$$

(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

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0, \tag{6}$$

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 $x_n = 1$. The step length or spacing is defined as $h = 1/n$. We have then the boundary conditions $v_0 = v_{n+1} = 0$. Now, where we would ordinarily approximate the second derivative with a three point formula, we will use a more sophisticated method. We will instead use the five point method

$$-v_i'' \approx -\frac{-v_{i-2} + 16v_{i-1} - 30v_i + 16v_{i+1} - v_{i+2}}{12h^2} = f_i \quad \text{for } i = 0, 1, 2, \ldots, n, \tag{7}$$

where $f_i = f(x_i)$. Please note that the local error of this five point method scales as $\mathcal{O}(h^4)$, whereas the corresponding local error of the three point method scales as $\mathcal{O}(h^2)$. We thus expect the five point method to result in significantly less error at the same number of points, n.

Show that you can rewrite this equation as a linear set of equations of the form

$$A\mathbf{v} = \tilde{\mathbf{b}}, \tag{8}$$

where $\tilde{b}_i = -12h^2 f_i$ and A is the $n \times n$ pentadiagonal matrix

$$A = \begin{bmatrix} & & & & & & & & \\ \ddots & & -1 & & & & & & \\ & -30 & 16 & -1 & & & & & \\ -1 & 16 & -30 & 16 & -1 & & & & \\ & -1 & 16 & -30 & 16 & -1 & & & \\ & & -1 & 16 & -30 & 16 & -1 & & \\ & & & -1 & 16 & -30 & 16 & -1 & \\ & & & & -1 & 16 & -30 & & \\ & & & & & -1 & & \ddots & \\ & & & & & & & & \end{bmatrix} \tag{9}$$

Recall that the vector $\mathbf{v}$ is the discretized solution vector. Note: Do *not* worry about the boundary conditions just yet, i.e. just consider the internal points $i = 2, 3, \ldots, n-3, n-2$.

In our case we will assume that the source term is $f(x) = 100 \exp(10x)$, and keep the same interval and boundary conditions. Then the above differential equation has a closed-form solution given by

$$u(x) = 1 - (1 \exp(-10))\, x - \exp(-10x).$$

Convince yourself that this is correct by inserting the solution in the Poisson equation, or solve the equation explicitly by integrating $u''(x)$ twice and use the boundary conditions to determine the resulting two integration constants. We will compare our numerical solution with this result in the next exercise.

We will now consider in some detail what happens near the boundaries, i.e. at $x = 0$ and $x = 1$. Since we know the solution at the points $x_0 = 0$ and $x_n = 1$, we may demand that our equation set adhere to this by inserting the equation

$$v_0 = 0. \tag{10}$$

This corresponds to setting the first row of the matrix A equal to $\begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \end{bmatrix}$, while also setting $\tilde{b}_0 = 0$.

Next, we need to figure out what to do about the $i = 1$ equation from Eq. (7), which states

$$-v_{-1} + 16v_{-0} - 30v_1 + 16v_2 - v_3 = -12h^2 f_0. \tag{11}$$

The reason why this is a problem is the presence of the v_{-1} term. According to our discretization from earlier, we only have v_i defined for points $i = 0, 1, 2, \ldots, n$. A common way of dealing with this extra term is to consider the derivative of v at the boundary. Consider the central difference approximation to the first derivative,

$$v'(x) \approx \frac{v(x+h) - v(x-h)}{h}. \tag{12}$$

(b) Use Eq. (12) at $x = 0$, in order to derive an expression for v_{-1} in terms of v_1, h. Use the extra boundary condition $u'(0) = 9 + e^{-10}$. Insert the expression for v_{-1} into Eq. (11) and insert the result into the second row of the matrix A. Remember to also modify the right hand side vector, $\tilde{\mathbf{b}}$, in the appropriate way. Note: Keep careful track of the signs!

(c) Perform the same steps at the right boundary, at $x = 1$. In other words, modify the last row of the matrix A and the last element of the right hand side vector, $\tilde{b}_n$, in the appropriate way. Also, set up the $i = n - 1$ version of Eq. (7), and use the derivative of u at $x = 1$ in order to eliminate the v_{n+1} element. Use the extra boundary condition $u'(1) = -1 + 11e^{-10}$.

(d) Having now set up the matrix fully, we are ready to derive the algorithm for solving matrix-vector equation. First, we can rewrite our matrix A in terms of one-dimensional vectors a, b, c, d, and e of length $n + 1$. Our linear equation reads

$$A\mathbf{v} = \begin{bmatrix} c_0 & d_0 & e_0 & & & & & \\ b_1 & c_1 & d_1 & e_1 & & & & \\ a_2 & b_2 & c_2 & d_2 & e_2 & & & \\ & a_3 & b_3 & c_3 & d_3 & e_3 & & \\ & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ & & & a_{n-2} & b_{n-2} & c_{n-2} & d_{n-2} & e_{n-2} \\ & & & & a_{n-1} & b_{n-1} & c_{n-1} & d_{n-1} \\ & & & & & a_n & b_n & c_n \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ \\ \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_0 \\ \tilde{b}_1 \\ \tilde{b}_3 \\ \vdots \\ \\ \\ \tilde{b}_n \end{bmatrix}. \tag{13}$$

Now, perform Gaussian elimination on the augmented matrix $\begin{bmatrix} A & \tilde{\mathbf{b}} \end{bmatrix}$ in order to arrive at an efficient algorithm for solving the pentadiagonal matrix system.

Hint: Start from the top and use the c_i element in row i to eliminate b_{i+1} and a_{i+2} in the subsequent two rows. Remember to account for what happens to the elements of the right hand side vector. After doing this all the way down to the nth row, you should have eliminated all the as and all the bs. You may need to handle the last b element, b_n, outside of the loop

Next, start from the nth row and work your way up, solving for v_i as you go. On row i, you will need the solution from the previous two rows, v_{i+1} and v_{i+2}.

(e) Set $n = 10$ and solve the matrix-vector equation you set up in exercises (a) through (c), using your newly developed algorithm. Test your implementation by comparing the solution to one obtained by a solver known to be correct (e.g. the built-in solve function in Armadillo, the \ function in Matlab, or the solve function in Numpy, etc.) If the two solutions differ by more than say 10^{-14}, your solver is not implemented correctly *or* the matrix A is not set up correctly.

(f) Use your solver to solve systems with n ranging from $n = 10$ to $n = 10^6$ (use at least 6 different n in this range), and compute the maximum value of the absolute error in your approximation for each n. Use the known exact solution, $u(x) = 1-(1-e^{-10})x-e^{-10x}$.

Plot the error as a function of n in a log-log plot. Comment on what you see. What is the slope of the curve in your log-log plot, and what does it tell you?

Note: Which n values you choose is up to you. The simplest possible way is to use $n = 10, 10^2, 10^3, 10^4, 10^5, 10^6$, but the plot will be easier to interpret if you choose a more sophisticated scheme and more points. Using N linearily spaced points between $n = 10$ and $n = 10^6$ might give a better result, but you may note that the data is very dense for low values of n and very sparse for the large values. The best alternative is to

use N logarithmically spaced points from $n = 10$ to $n = 10^6$. Using $N = 50$ should be sufficient.

(g) Calculate how the absolute error, $\max_{x\in(0,1)} |u(x) - v(x)|$ scales as a function of the step length, h. Remember that the five point approximation we use for the second derivative has an error scaling on the order of $\mathcal{O}(h^4)$. Is the error you find in line with what you would expect, given this? If not, can you offer an explanation as to why we dont see this behaviour?