

Project 1

Due date **19.th of September, 2016 - 23:59**

Øyvind B. Svendsen, Magnus Christopher Bareid
un: oyvinbsv, magnucb

September 18, 2016

Abstract

The aim of this project is for the students 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.

The students were invited to use either brute force-algorithms to calculate linear algebra, or to use a set of recommended linear algebra packages, i.e. Armadillo, that simplify the syntax of linear algebra. Additionally, dynamic memory handling is expected.

The students will showcase necessary algebra to perform the tasks given to them, and explain the way said algebra is implemented into algorithms. In essence, we're asked to simplify a linear second-order differential equation from the form of the Poisson equation, seen as

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

into a one-dimensional form bounded by Dirichlet boundary conditions.

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

so that discretized linear algebra may be committed unto the equation, and yield a method for numerical integration.

Contents

1	Computational Physics, first project	3
1.1	The fundamental math	3
1.1.1	Introduction	3
1.1.2	Problem	3
1.1.3	Method	3

1 Computational Physics, first project

1.1 The fundamental math

1.1.1 Introduction

The production of this document will inevitably familiarize its authors with the programming language C++, and to this end mathematical groundwork must first be elaborated to translate a Poisson equation from continuous calculus form, into a discretized numerical form.

The Poisson equation is rewritten to a simplified form, for which a real solution is given, with which we will compare our numerical approximation to the real solution.

The real motivation for this project, though, is to find a viable numerical method for double integration and the implementation of such.

1.1.2 Problem

1.1.3 Method

Reviewing the Poisson equation:

$$\begin{aligned}\nabla^2 \Phi &= -4\pi\rho(\mathbf{r}), \text{ which is simplified one-dimensionally by } \Phi(r) = \phi(r)/r \\ \Rightarrow \frac{d^2\phi}{dr^2} &= -4\pi r\rho(r), \text{ which is further simplified by these substitutions:} \\ r &\rightarrow x, \\ \phi &\rightarrow u, \\ 4\pi r\rho(r) &\rightarrow f, \quad \text{which produces the simplified form}\end{aligned}$$

$$\begin{aligned}-u''(x) &= f(x), \quad \text{for which we assume that } f(x) = 100e^{-10x}, \\ \Rightarrow u(x) &= 1 - (1 - e^{-10})x - e^{-10x}, \text{ with bounds: } x \in [0, 1], \quad u(0) = u(1) = 0\end{aligned}\tag{1}$$

From here on and out, the methods for finding the doubly integrated function $u(x)$ numerically will be deduced.

To more easily comprehend the syntax from a programming viewpoint, one may refer to the each discretized representation of x and u ; we know the span of x , and therefore we may divide it up into appropriate chunks. Each of these x_i will yield a corresponding u_i .

We may calculate each discrete x_i by the form $x_i = ih$ in the interval from $x_0 = 0$ to $x_{n+1} = 1$ as it is linearly increasing, meaning we use $n + 1$ points in our approximation, yielding the step length $h = 1/(n + 1)$. Of course, this also yields the discretized representation of $u(x_i) = u_i$.

Through Euler's teachings on discretized numerical derivation methods, a second derivative may be constructed through the form of

$$\begin{aligned}\left(\frac{\partial}{\partial x}\right)_{fw} u(x) &= \frac{u_{i+1} - u_i}{h}, \quad \left(\frac{\partial}{\partial x}\right)_{bw} u(x) = \frac{u_i - u_{i-1}}{h}, \\ \left(\frac{\partial}{\partial x}\right)^2 u_i &= \left(\frac{\partial}{\partial x}\right)_{bw} \left(\frac{\partial}{\partial x}\right)_{fw} u_i = \left(\frac{\partial}{\partial x}\right)_{bw} \left(\frac{u_{i+1} - u_i}{h}\right) = \frac{\left(\frac{\partial}{\partial x}\right)_{bw} u_{i+1} - \left(\frac{\partial}{\partial x}\right)_{bw} u_i}{h} \\ \left(\frac{\partial}{\partial x}\right)^2 u_i &= \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}, \quad \text{which we then use for the problem in question, (1).} \\ \Rightarrow -\left(\frac{\partial}{\partial x}\right)^2 u(x) &= -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = \frac{2u_i - u_{i+1} - u_{i-1}}{h^2} = f_i, \quad \text{for } i = 1, \dots, n\end{aligned}\tag{2}$$

The discretized problem can now be solved as a linear algebra problem. Looking closer at the discretized problem:

$$\begin{aligned} -u''(x_i) &= \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} = f_i, & \text{for } i = 1, \dots, n. \\ \Rightarrow & -u_{i+1} + 2u_i - u_{i-1} = h^2 f_i, & \text{substitute } h^2 f_i = y_i, \text{ and test for some values:} \end{aligned}$$

$$\begin{aligned} i = 1 : & \quad -u_2 + 2u_1 - u_0 = y_1 \\ i = 2 : & \quad -u_3 + 2u_2 - u_1 = y_2 \\ i = 3 : & \quad -u_4 + 2u_3 - u_2 = y_3 \\ & \quad \vdots \\ i = n : & \quad -u_{n+1} + 2u_n - u_{n-1} = y_n \end{aligned}$$

By now it should be obvious to recognize that the coefficients corresponding to each of these terms and their corresponding values of $u(x)$ looks very similar to a tridiagonal matrix multiplication problem which could be represented such as this:

$$-\left(\frac{\partial}{\partial x}\right)^2 u(x) = f(x) \quad \Rightarrow \quad \hat{\mathbf{A}}\hat{\mathbf{u}} = \hat{\mathbf{y}} \quad \Rightarrow \quad \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_{n+1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n+1} \end{pmatrix}$$

This matrix equation will not be valid for the first, and last, values of $\hat{\mathbf{y}}$ because they would require elements of $\hat{\mathbf{u}}$ that are not defined; u_{-1} and u_{n+1} , respectively. Given this constraint we see that the matrix-equation gives the same set of equations that we require.

$$\begin{aligned} i = 1 : & \quad -u_2 + 2u_1 - u_0 = y_1 \\ i = 2 : & \quad -u_3 + 2u_2 - u_1 = y_2 \\ i = 3 : & \quad -u_4 + 2u_3 - u_2 = y_3 \\ & \quad \vdots \\ i = n : & \quad -u_{n+1} + 2u_n - u_{n-1} = y_n \end{aligned}$$

The original problem at hand (the simplified Poisson equation) has now been reduced to a numerical linear algebra problem. Solving a tridiagonal matrix-problem like this is done by Gaussian elimination of the tridiagonal matrix $\hat{\mathbf{A}}$, and thereby solving $\hat{\mathbf{u}}$ for the resulting diagonal-matrix, as presumably $\hat{\mathbf{A}}$ and $\hat{\mathbf{y}}$ are the knowns in this set.

Firstly the tridiagonal matrix $\hat{\mathbf{A}}$ is rewritten to a series of three vectors $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{c}}$ that will represent a general tridiagonal matrix. This will make it easier to include other problems of a general form later.

The tridiagonal matrix $\hat{\mathbf{A}}$ with the vector $\hat{\mathbf{y}}$ included now looks like:

$$\begin{pmatrix} b_0 & c_0 & 0 & \dots & \dots & 0 & y_0 \\ a_1 & b_1 & c_1 & \ddots & & \vdots & \vdots \\ 0 & a_2 & b_2 & c_2 & \ddots & \vdots & \vdots \\ \vdots & \ddots & a_3 & b_3 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_n & \vdots \\ 0 & \dots & \dots & 0 & a_{n+1} & b_{n+1} & y_{n+1} \end{pmatrix}$$

, but we only work with rows from $i = 1$ to $i = n$ because of the Dirichlet conditions, as explained above. The matrix which we row-reduce thus looks like this

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 & y_1 \\ a_2 & b_2 & c_2 & \ddots & & \vdots & \vdots \\ 0 & a_3 & b_3 & c_3 & \ddots & \vdots & \vdots \\ \vdots & \ddots & a_4 & b_4 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_n & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n & y_n \end{pmatrix}$$

The Gaussian elimination can be split into two parts; a forward substitution where the matrix-elements a_i are set to zero, and a backward substitution where the vector-elements u_i are calculated from known values.

Starting the Gaussian elimination with the second row, row II, a row operation is performed to maintain the validity of the system. The goal is to remove element a_2 from the row. This is done by subtracting row II, multiplied with some constant k from row I. For every next row operation, there will then be a new k_i calculated.

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 & y_1 \\ \tilde{a}_2 & \tilde{b}_2 & \tilde{c}_2 & \ddots & & \vdots & \tilde{y}_2 \\ 0 & a_3 & b_3 & c_3 & \ddots & \vdots & \vdots \\ \vdots & \ddots & a_4 & b_4 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_n & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n & y_n \end{pmatrix}$$

$$\tilde{\Pi} = \Pi - k_I \times \text{I}$$

where k_I is determined by

$$\tilde{a}_2 = 0 = a_2 - k_I b_1 \Rightarrow k_I = \frac{a_2}{b_1}$$

$$\tilde{b}_2 = b_2 - \frac{a_2}{b_1} c_1$$

$$\tilde{c}_2 = c_2 - \frac{a_2}{b_1} \times 0 = c_2$$

$$\tilde{y}_2 = y_2 - \frac{a_2}{b_1} y_1$$

Moving on to row 3, and performing a similar operation:

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & 0 & y_1 \\ 0 & \tilde{b}_2 & c_2 & \ddots & & \vdots & \tilde{y}_2 \\ 0 & \tilde{a}_3 & \tilde{b}_3 & \tilde{c}_3 & \ddots & \vdots & \tilde{y}_3 \\ \vdots & \ddots & a_4 & b_4 & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \ddots & c_n & \vdots \\ 0 & \dots & \dots & 0 & a_n & b_n & y_n \end{pmatrix}$$

$$\begin{aligned}
& \tilde{\text{III}} = \text{II} - k_{\text{II}} \times \text{II} \\
& \text{where } k_{\text{II}} \text{ is determined by} \\
& \tilde{a}_3 = 0 = a_3 - k_{\text{II}} \tilde{b}_2 \quad \Rightarrow k_{\text{II}} = \frac{a_3}{\tilde{b}_2} \\
& \tilde{b}_3 = b_3 - \frac{a_3}{\tilde{b}_2} c_2 \\
& \tilde{c}_3 = c_3 - \frac{a_3}{\tilde{b}_2} \times 0 = c_3 \\
& \tilde{y}_3 = y_3 - \frac{a_3}{\tilde{b}_2} \tilde{y}_2
\end{aligned}$$

Having repeated this procedure, a pattern emerges and an algorithm can be formulated:

$$\begin{aligned}
\tilde{b}_{i+1} &= b_{i+1} - \frac{a_{i+1}}{\tilde{b}_i} c_i \\
\tilde{y}_{i+1} &= y_{i+1} - \frac{a_{i+1}}{\tilde{b}_i} \tilde{y}_i \\
i &= 1, 2, \dots, n-1
\end{aligned}$$

After this procedure, the tridiagonal matrix $\hat{\mathbf{A}}$ is transformed into an uppertriangular matrix. This sort of set of equations can be solved iteratively for $\hat{\mathbf{u}}$, since the last equation has one unknown and the other equations has only two unknowns.

2 Appendix - Program list

This is the code used in this assignment. Anything that was done by hand has been implemented into this pdf, above. `plot_stuff.py`

```
1 import pylab as pyl
2 import os
3 import sys
4
5 curdir = os.getcwd()
6 data_dict = {} #dictionary of files
7 n_range = [10,50,80, 100, 500, 800, 1000, 5000, 10000]
8
9 for n in n_range:
10     #loop through different n's
11     with open(curdir+"/data/dderiv_u_c++_n%d_tridiag.dat"%(n), 'r') as infile:
12         full_file = infile.read() #read entire file into text
13         lines = full_file.split('\n') #separate by EOL-characters
14         lines = lines[:-1] #remove last line (empty line)
15         keys = lines.pop(0).split(',') #use top line as keys for dictionary
16         dict_of_content = {}
17         for i, key in zip(range(len(keys)), keys): #loop over keys, create approp. arrays
18             dict_of_content[key] = [] #empty list
19             for j in range(len(lines)): #loop through the remaining lines of the data-set
20                 line = lines[j].split(',') #split the line into string-lists
21                 word = line[i] # append the correct value to the correct list with the correct key
22                 try:
23                     word = float(word) #check if value can be float
24                 except ValueError: #word cannot be turned to number
25                     print word
26                     sys.exit("There is something wrong with your data-file \n'%s' cannot be turned to numbers"%word)
27                 dict_of_content[key].append(word)
28             data_dict["n=%d"%n] = dict_of_content #add complete dictionary to dictionary of files
29
30 def u_exact(x):
31     u = 1.0 - (1.0 - pyl.exp(-10.0))*x - pyl.exp(-10.0*x)
32     return u
33
34 def plot_generator(version, n):
35     """
36     plot generator of generated data
37     """
38     datafile = open(curdir+"/data/dderiv_u_python_v%s_n%d.dat"%(version,n))
39     data = []
40
41     for line in datafile:
42         linesplit = [item.replace(",","") for item in line.split()]
43         data.append(linesplit)
44
45     columns = data[0]
46     data = pyl.array(data[1:]).astype(pyl.float64)
47     for i in xrange(len(columns)-1):
48         pyl.figure() # comment out this line to unify the plots ... when their dimensions correlate
49         pyl.plot(data[:,0], data[:,i+1], label=r"%s" % columns[i+1])
50         pyl.xlabel("x")
51         pyl.ylabel(r"%s" % columns[i+1])
52         pyl.title(r"Plot of %s over x" % columns[i+1])
53         pyl.legend(loc='best')
54
55     # pyl.savefig("evil_plot.png", dpi=400)
```

```

56     pyl.show()
57     datafile.close()
58
59 def harry_plotter():
60     #plot pre-game
61     pyl.figure()
62     pyl.grid(True)
63     pyl.title("function u for different steplengths")
64     pyl.ylabel("u(x)")
65     pyl.xlabel("x")
66     for n in n_range:
67         x = pyl.array(data_dict["n=%d"%n]["x"])
68         u_gen = pyl.array(data_dict["n=%d"%n]["u_gen"])
69         u_spec = pyl.array(data_dict["n=%d"%n]["u_spec"])
70         u_LU = pyl.array(data_dict["n=%d"%n]["u_LU"])
71         pyl.plot(x,u_gen, 'g-', label="general tridiag, n=%d"%n)
72         pyl.plot(x,u_spec, 'r-', label="specific tridiag, n=%d"%n)
73         pyl.plot(x,u_LU, 'b-', label="LU-dekomp, n=%d"%n)
74
75     u_ex = u_exact(x)
76     pyl.plot(x, u_ex, 'k-', label="exact, n=%d"%len(x))
77     #pyl.legend(loc="best",prop={"size":8})
78     pyl.show()
79     return None
80
81 def compare_methods(n):
82     """
83     For a specific length 'n' compare both methods
84     with the exact function.
85     """
86     x = pyl.array(data_dict["n=%d"%n]["x"])
87     gen = pyl.array(data_dict["n=%d"%n]["u_gen"])
88     spec = pyl.array(data_dict["n=%d"%n]["u_spec"])
89     exact = u_exact(x)
90     pyl.figure("compare methods")
91     pyl.grid(True)
92     pyl.hold(True)
93     pyl.xlabel("x")
94     pyl.ylabel("u(x)")
95     pyl.title("function u for three different methods (n=%d)"%n)
96
97     pyl.plot(x, exact, 'k-', label="exact")
98     pyl.plot(x, gen, 'b--', label="general tridiagonal")
99     pyl.plot(x, spec, 'g-', label="specific tridiagonal")
100    pyl.legend(loc='best', prop={'size':9})
101    pyl.savefig(curdir+"/img/compare_methods_n%d.png"%n)
102
103 def compare_approx_n(n_range=[10,100,1000], approx_string="general"):
104     """
105     For all n's available, plot the general approximation and
106     exact solution
107     """
108     if approx_string == "general":
109         approx_key = "u_gen"
110     elif approx_string == "specific":
111         approx_key = "u_spec"
112     else:
113         sys.exit("In function 'compare_approx_n', wrong argument 'approx_string'")
114     pyl.figure("compare %s"%approx_string)
115     pyl.grid(True)

```



```

116     pyl.hold(True)
117     pyl.xlabel("x")
118     pyl.ylabel("u(x)")
119     pyl.title("approximation by %s tridiagonal method"%approx_string)
120
121     for n in n_range:
122         x = pyl.array(data_dict["n=%d"%n]["x"])
123         u_approx = pyl.array(data_dict["n=%d"%n][approx_key])
124         pyl.plot(x, u_approx, '--', label="n=%1.1e"%n)
125
126     x = pyl.linspace(0,1,1001)
127     exact = u_exact(x)
128     pyl.plot(x, exact, '-', label="exact")
129     pyl.legend(loc='best', prop={'size':9})
130     pyl.savefig(curdir+"/img/compare_%s_n_%d.png"%(approx_string,n))
131
132 def epsilon_plots(n_range=[10,100,1000]):
133     eps_max = pyl.zeros(len(n_range))
134     h = pyl.zeros(len(n_range))
135     for i, n in enumerate(n_range):
136         x = pyl.array(data_dict["n=%d"%n]["x"])
137         u = u_exact(x)
138         v = pyl.array(data_dict["n=%d"%n]["u_gen"])
139         #calculate eps_max by finding max of |v_i - u_i|
140         max_diff_uv = 0; jmax = 0;
141         for j in range(n):
142             diff_uv = abs(v[j]-u[j])
143             if diff_uv > max_diff_uv:
144                 max_diff_uv = diff_uv
145                 jmax = j
146         if jmax == 0 or jmax == n-1:
147             sys.exit("There is an error in calculating the max_epsilon")
148         eps_max[i] = pyl.log10(max_diff_uv/float(abs(u[jmax])))
149         h[i] = pyl.log10(1.0/(n+1))
150     pyl.figure("epsilon")
151     pyl.grid(True)
152     pyl.hold(True)
153     pyl.xlabel(r"\log_{10}(h)")
154     pyl.ylabel(r"$\epsilon = \log_{10}(\frac{u_{approx}-u_{exact}}{u_{exact}})$")
155     pyl.title("log-plot of epsilon against step-length h")
156     pyl.plot(h, eps_max, 'ko')
157     pyl.legend(loc='best')
158     pyl.savefig(curdir+"/img/epsilon.png")
159
160 #make plots
161 compare_methods(n=10)
162 #compare_approx_n(approx_string="general")
163 #compare_approx_n(approx_string="specific")
164 pyl.show()
165 #epsilon_plots()
166 #pyl.show()

```