

# 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 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 student was invited to use either brute force-algorithms to calculate linear algebra, or to use a set of recommended linear algebra packages through 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, yielding a number of numerical methods for acquiring the underivated function  $u(x)$ .

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Problem</b>	<b>3</b>
<b>3</b>	<b>Method</b>	<b>3</b>

<b>4</b>	<b>Explanation of programs</b>	<b>6</b>
4.1	main.cpp . . . . .	6
4.2	make_data.py . . . . .	7
4.3	plot_stuff.py . . . . .	8
<b>5</b>	<b>Results</b>	<b>8</b>
5.1	General tridiagonal solver . . . . .	8
5.2	CPU-time . . . . .	8
<b>6</b>	<b>Appendix - Github</b>	<b>9</b>

# 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.

## 2 Problem

## 3 Method

Reviewing the Poisson equation:

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

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

From here on and out, the methods for finding  $u(x)$  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  $x_n = 1$  as it is linearly increasing, meaning we use  $n$  points in our approximation, yielding the step length  $h = 1/n$ . Of course, this also yields 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

$$\begin{aligned} \left(\frac{\partial u}{\partial x}\right)_{fw} &= \frac{u_{i+1} - u_i}{h} & \left(\frac{\partial u}{\partial x}\right)_{bw} &= \frac{u_i - u_{i-1}}{h} \\ \left(\frac{\partial}{\partial x}\right)^2 [u_i] &= \left(\frac{\partial}{\partial x_{bw}}\right) \left(\frac{\partial}{\partial x}\right)_{fw} [u_i] = \left(\frac{\partial}{\partial x}\right)_{bw} \left(\frac{u_{i+1} - u_i}{h}\right) = \frac{\left(\frac{\partial u_{i+1}}{\partial x}\right)_{bw} - \left(\frac{\partial u_i}{\partial x}\right)_{bw}}{h} \\ &= \left(\frac{\partial}{\partial x}\right)^2 [u_i] = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} \\ -u''(x) &= -\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} = f_i, \quad \text{for } i = 1, \dots, n \end{aligned} \quad (2)$$

The discretized problem can now be solved as a linear algebraic problem. Looking closer at

the discretized problem:

$$\begin{aligned}
-u''(x_i) &= \frac{-u_{i+1} + 2u_i - u_{i-1}}{h^2} = f_i \\
&\Rightarrow -u_{i+1} + 2u_i - u_{i-1} = h^2 f_i = y_i \\
i = 1 : \quad &-u_2 + 2u_1 - u_0 = y_1 \\
i = 2 : \quad &-u_3 + 2u_2 - u_1 = y_2 \\
i = 3 : \quad &-u_4 + 2u_3 - u_2 = y_3 \\
&\vdots \\
i = n : \quad &-u_{n+1} + 2u_n - u_{n-1} = y_n
\end{aligned}$$

This is very similar to a linear algebra / matrix problem and we will test a system of equations to match.

$$A\vec{u} = \vec{y}$$

$$\begin{bmatrix} 2 & -1 & 0 & \dots \\ -1 & 2 & -1 & \dots \\ 0 & -1 & 2 & \ddots \\ \vdots & \vdots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{n+1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n+1} \end{bmatrix}$$

This matrix equation will not be valid for the first and last values of  $\vec{y}$  because they would require elements of  $\vec{u}$  that are not defined;  $u_{-1}$  and  $u_{n+2}$ . Given this constraint we see that the matrix-equation gives the same set of equations that we require.

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

The coefficients from each of these terms and their corresponding value of  $u(x)$  may be represented by a tridiagonal matrix multiplication:

$$-\frac{d^2}{dx^2}u(x) = f(x) \quad \Rightarrow \quad \hat{\mathbf{A}}\hat{\mathbf{u}} = h^2\hat{\mathbf{f}} \quad \Rightarrow \quad \begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & & \vdots \\ 0 & -1 & 2 & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & \ddots & -1 \\ 0 & \dots & \dots & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} u_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_n \end{pmatrix} = h^2 \begin{pmatrix} f_0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ f_n \end{pmatrix}$$

The double derivation is now reduced to a discretized linear algebra operation by way of matrix multiplication. In our case,  $f(x)$  is known to us, and the only unknowns are the  $u(x)$ 's from  $u_1 \rightarrow u_{n-1}$ , as per the Dirichlet boundary conditions, which allows the use of the algorithm from equation .

The original problem at hand (the Poisson equation) has now been "degraded" to a simpler, linear algebra problem.

Solving a tridiagonal matrix-problem like this is done by gaussian elimination of the tridiagonal matrix A, and thereby solving  $\vec{u}$  for the resulting diagonal-matrix.

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

The tridiagonal matrix A (with the vector y) now looks like:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & y_1 \\ a_2 & b_2 & c_2 & 0 & & y_2 \\ 0 & a_3 & b_3 & c_3 & & \vdots \\ 0 & 0 & a_4 & b_4 & \ddots & \\ \vdots & & & \ddots & \ddots & c_{n-1} & y_{n-1} \\ & & & a_n & b_n & y_n \end{bmatrix}$$

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 with row 2, 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 1 (multiplied with some constant 'k' from row 2.

$$\begin{aligned} \tilde{Row}_2 &= Row_2 - k \times Row_1 \\ \begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & y_1 \\ \tilde{a}_2 & \tilde{b}_2 & \tilde{c}_2 & 0 & & \tilde{y}_2 \\ 0 & a_3 & b_3 & c_3 & & y_3 \\ 0 & 0 & a_4 & b_4 & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & c_{n-1} & y_{n-1} \\ & & & a_n & b_n & y_n \end{bmatrix} & \text{where } k \text{ is determined by } \tilde{a}_2 = 0 \Rightarrow k = \frac{a_2}{b_1} \\ \tilde{b}_2 &= b_2 - \frac{a_2}{b_1} c_1 \\ \tilde{c}_2 &= c_2 - \frac{a_2}{b_1} \times 0 = c_2 \\ \tilde{y}_2 &= y_2 - \frac{a_2}{b_1} y_1 \end{aligned}$$

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

$$\begin{aligned} \tilde{Row}_3 &= Row_3 - k \times Row_2 \\ \begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & y_1 \\ 0 & \tilde{b}_2 & \tilde{c}_2 & 0 & & \tilde{y}_2 \\ 0 & \tilde{a}_3 & \tilde{b}_3 & \tilde{c}_3 & & \tilde{y}_3 \\ 0 & 0 & a_4 & b_4 & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & c_{n-1} & y_{n-1} \\ & & & a_n & b_n & y_n \end{bmatrix} & \text{where } k \text{ is determined by } \tilde{a}_3 = 0 \Rightarrow k = \frac{a_3}{\tilde{b}_2} \\ \tilde{b}_3 &= b_3 - \frac{a_3}{\tilde{b}_2} \tilde{c}_2 \\ \tilde{c}_3 &= c_3 - \frac{a_3}{\tilde{b}_2} \times 0 = c_3 \\ \tilde{y}_3 &= y_3 - \frac{a_3}{\tilde{b}_2} \tilde{y}_2 \end{aligned}$$

By repeating this step a pattern emerges, and an algorithm can be found:

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

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

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & y_1 \\ 0 & \tilde{b}_2 & c_2 & 0 & & \tilde{y}_2 \\ 0 & 0 & \tilde{b}_3 & \tilde{c}_3 & & \tilde{y}_3 \\ 0 & 0 & a_4 & b_4 & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & c_{n-1} & y_{n-1} \\ & & & & a_n & b_n & y_n \end{bmatrix}$$

## 4 Explanation of programs

### 4.1 main.cpp

The main-program is a c++-program designed to take a cmd-line argument that decides the size of the array u, and a boolean argument (0 or 1 that decides wether or not the armadillo-function "solve" should be used.

An x-array between 0 and 1 is calculated and the appropriate a,b,c,y-arrays are calculated as well. For this program the tridiagonal elements are constantly -1, 2 and -1 while y(x) follows the function described in the introduction.

If the boolean cmd-line argument is false, then the program will calculate  $\vec{u}$  using the general tridiagonal method(explanation below) and store the CPU-time it takes to calculate  $\vec{u}$  using this algorithm. Next,  $\vec{u}$  will be calculated using the specialized tridiagonal method(explanation below) and store the CPU-time it takes to calculate  $\vec{u}$  using this algorithm.

Afterwards, since c++ is terrible at plotting data, all the data (CPU-time of methods, x-arrays, and u-arrays for both methods) are stored in separate files in the data-folder with the csv-format.

If the boolean cmd-line argument is true, then the program will calculate  $\vec{u}$  using LU-decomposition method in the armadillo-library, measure the CPU-time it takes and store this time in the time-datafile.

The function 'write2file' does exactly what the name suggest, write a string to a file. The file-name is a argument to the function and the string is also a argument to the function. It is worth noting that this function will only append to the end of a file so that it does not accidentally destroy lots of data.

The function 'general\_tridiag' solves the equation  $A\hat{x} = \hat{y}$  for  $\hat{x}$  when A is a tri-diagonal matrix. The tridiagonal elements are three arrays of length n that must be given to the function

```

1 t0 = clock();
2 //forward substitution
3 for (int i=1; i<=arg_n-3; i++){
4     k = arg_a(i+1)/((double) arg_b(i)); //1 flop
5     arg_b(i+1) -= k*arg_c(i); //2 flops
6     arg_y(i+1) -= k*arg_y(i); //2 flops
7 }
8 //backward substitution
9 for (int i=arg_n-2; i>=1; i--){
10     arg_u(i) = (arg_y(i) - arg_u(i+1)*arg_c(i))/((double) arg_b(i)); //4 flops
11 }
12 t1 = clock();
13 return (t1 - t0)/((double) CLOCKS_PER_SEC); //measure time of forward and backward substitution

```

Figure 1: The syntax used in the function 'general\_tridiag' where all variables beginning with arg are function arguments.

as arguments along with the vector  $y$  and the size of the arrays. In figure ?? the syntax for solving  $x_i$  is presented, calculating  $\tilde{b}_i$  and  $\tilde{y}_i$  in the forward substitution, and  $u_i$  in backward substitution, according to the algorithm in section 3.

The comments in figure ?? counts the number of floating point operations in each line. This amounts to

$$5flops \times (n - 3)iterations + 4flops \times (n - 2)iterations = (9n - 23)flops$$

This strange number comes from the fact that the program requires  $n$  to be at least 3, since the Dirichlet Boundary Conditions are included in the for-loops. (i.e. excluding calculation of the endpoints, leaving them to be zero according to DBC).

The function 'specific\_tridiag' is an attempt at optimizing the code in 'general\_tridiag' and lowering the number of floating point operations per for-loop. This is primarily done by inserting  $-1.0$  for every tridiagonal element  $a_i$  and  $c_i$ , and precalculating the diagonal elements  $d_i$  using the formula  $d_i = \frac{i+1}{i}$ .

In figure ?? the algorithm for an optimized, special-case of the tri-diagonal solver is included. The diagonal elements are already calculated, meaning the forward substitution only applies to  $\hat{y}$ . The comments counts the number of floating point operations in each iteration.

$$2flops \times (n - 3)iterations + 2flops \times (n - 2)iterations = (4n - 10)flops$$

The same restraint applies to this calculation, meaning since the algorithm includes the DBC,  $n$  must be three or higher to actually compute any values for  $\hat{u}$ .

The function 'LU-decomp' simply solves the set of equations by making the matrix  $A$  with armadillo-arrays and using the solve function in the armadillo-library. the function returns the CPU-time it takes to calculate  $\hat{u}$

## 4.2 make\_data.py

The program performs the experiment(running 'main.cpp') for several values of  $n$ .

Firstly it calculates both tridiagonal solvers and LU-decomposition for  $n = 10, 100, 1000$  in order to compare the CPU-time for the three methods.

Then it chooses a logarithmic scale of  $n$ -values between  $n=10$  and  $n=10'000$  in order to produce data-sets for plotting  $u(x)$  and  $\epsilon$ .

```

1 t0 = clock();
2 //forward substitution
3 for (int i=2; i<=arg_n-2; i++){
4     arg_y(i) += arg_y(i-1)/d(i-1); //2 flops
5 }
6 //backward substitution
7 for (int i=arg_n-2; i>=1; i--){
8     arg_u(i) = (arg_y(i) + arg_u(i+1))/d(i); //2 flops
9 }
10 t1 = clock();
11 return (t1 - t0)/((double) CLOCKS_PER_SEC); //measure time of forward and backward substitution

```

Figure 2: The syntax used in the function 'specific\_tridiag' where all variables beginning with arg are function arguments.

Figure 3: Compare calculated  $u(x)$  with exact solution of  $u(x)$  for three values of  $n$ , using the general tridiagonal solver.

### 4.3 plot\_stuff.py

This program fetches relevant data produced by 'main.cpp'/'make\_data.py' and stores these data as arrays in a nested dictionary.

Three plotting functions are defined for making appropriate plots:

'compare\_methods' plots  $u(x)$  against  $x$  for both tridiagonal solvers and the exact solution for a specific  $n$ .

'compare\_approx\_n' plots  $u(x)$  against  $x$  for several values of  $n$  for one of the tri-diagonal solvers.

'epsilon\_plots' calculates the maximum relative error and makes a log-log-plot against step-length. This done by calculating the maximum difference between  $u_{exact}$  and  $u_{general}$  for a series of  $n$ -values, using this difference to find the relative error  $10^\epsilon = \left| \frac{v-u}{u} \right|$  for every  $n$ -value.

These functions are called to make the plots in section 5

## 5 Results

The data can be found in the data-folder on the github-repository<sup>1</sup> and all the figures presented in this section can found in the img-folder in the same repository.

### 5.1 General tridiagonal solver

### 5.2 CPU-time

---

<sup>1</sup>see section 6



$\log_{10}(n)$	general tridiagonal[s]	specific tridiagonal[s]	LU decomposition [s]
1	0.000010	0.000006	0.008585
2	0.000047	0.000007	0.015610
3	0.000159	0.000123	0.699342
4	0.001421	0.001749	
5	0.020054	0.024508	
6	0.056499	0.064189	

Table 1: CPU-time each methods use for calculating  $u(x)$  n seconds

## 6 Appendix - Github

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_LU = pyl.logspace(1,3,num=3)
8 n_range_tridiag = pyl.logspace(1, 4,num=7)
9
10 for n in n_range_tridiag:
11     #loop through different n's
12     with open(curdir+"/data/dderiv_u_c++_n%d_tridiag.dat"%(int(n)), 'r') as infile:
13         full_file = infile.read() #read entire file into text
14         lines = full_file.split('\n') #separate by EOL-characters
15         lines = lines[:-1] #remove last line (empty line)
16         keys = lines.pop(0).split(',') #use top line as keys for dictionary
17         dict_of_content = {}
18         for i, key in zip(range(len(keys)), keys): #loop over keys, create approp. arrays
19             dict_of_content[key] = [] #empty list
20             for j in range(len(lines)): #loop though the remaining lines of the data-set
21                 line = lines[j].split(',') #split the line into string-lists
22                 word = line[i] # append the correct value to the correct list with the correct key
23                 try:
24                     word = float(word) #check if value can be float
25                 except ValueError: #word cannot be turned to number
26                     print word
27                     sys.exit("There is something wrong with your data-file \n'%s' cannot be turned to numbers"%word)
28                 dict_of_content[key].append(word)
29             data_dict["n=%d"%n] = dict_of_content #add complete dictionary to dictionary of files
30
31 def u_exact(x):
32     u = 1.0 - (1.0 - pyl.exp(-10.0))*x - pyl.exp(-10.0*x)
33     return u
34
35 def compare_methods(n):
36     """
37     For a specific length 'n' compare both methods
38     with the exact function.
39     """
40     x = pyl.array(data_dict["n=%d"%n]["x"])

```

```

41 gen = pyl.array(data_dict["n=%d"%n]["u_gen"])
42 spec = pyl.array(data_dict["n=%d"%n]["u_spec"])
43 exact = u_exact(x)
44 pyl.figure("compare methods")
45 pyl.grid(True)
46 pyl.hold(True)
47 pyl.xlabel("x")
48 pyl.ylabel("u(x)")
49 pyl.title("function u for three different methods (n=%d)"%n)
50
51 pyl.plot(x, exact, 'k-', label="exact")
52 pyl.plot(x, gen, 'b--', label="general tridiagonal")
53 pyl.plot(x, spec, 'g-', label="specific tridiagonal")
54 pyl.legend(loc='best', prop={'size':9})
55 pyl.savefig(curdir+"/img/compare_methods_n%d.png"%n)
56
57 def compare_approx_n(n_range=[10,100,1000], approx_string="general"):
58     """
59     For all n's available, plot the general approximation and
60     exact solution
61     """
62     if approx_string == "general":
63         approx_key = "u_gen"
64     elif approx_string == "specific":
65         approx_key = "u_spec"
66     else:
67         sys.exit("In function 'compare_approx_n', wrong argument 'approx_string'")
68     pyl.figure("compare %s"%approx_string)
69     pyl.grid(True)
70     pyl.hold(True)
71     pyl.xlabel("x")
72     pyl.ylabel("u(x)")
73     pyl.title("approximation by %s tridiagonal method"%approx_string)
74
75     for n in n_range:
76         n = int(n)
77         x = pyl.array(data_dict["n=%d"%n]["x"])
78         u_approx = pyl.array(data_dict["n=%d"%n][approx_key])
79         pyl.plot(x, u_approx, '--', label="n=%1.1e"%n)
80
81     x = pyl.linspace(0,1,1001)
82     exact = u_exact(x)
83     pyl.plot(x, exact, '-', label="exact")
84     pyl.legend(loc='best', prop={'size':9})
85     pyl.savefig(curdir+"/img/compare_%s_exact.png"%(approx_string))
86
87 def epsilon_plots(n_range=[10,100,1000]):
88     eps_max = pyl.zeros(len(n_range))
89     h = pyl.zeros(len(n_range))
90     for i, n in enumerate(n_range):
91         x = pyl.array(data_dict["n=%d"%n]["x"])
92         u = u_exact(x)
93         v = pyl.array(data_dict["n=%d"%n]["u_gen"])
94         #calculate eps_max by finding max of |v_i - u_i|
95         max_diff_uv = 0; jmax = 0;
96         for j in range(n):
97             diff_uv = abs(v[j]-u[j])
98             if diff_uv > max_diff_uv:
99                 max_diff_uv = diff_uv
100             jmax = j

```

```

101         if jmax == 0 or jmax == n-1:
102             sys.exit("There is an error in calculating the max_epsilon")
103             eps_max[i] = pyl.log10(max_diff_uv/float(abs(u[jmax])))
104             h[i] = pyl.log10(1.0/(n+1))
105     pyl.figure("epsilon")
106     pyl.grid(True)
107     pyl.hold(True)
108     pyl.xlabel(r"\log_{10}(h)")
109     pyl.ylabel(r"$\epsilon = \log_{10}\left(\frac{u_{\text{approx}}-e_{\text{exact}}}{u_{\text{exact}}}\right)$")
110     pyl.title("log-plot of epsilon against step-length h")
111     pyl.plot(h, eps_max, 'ko')
112     pyl.legend(loc='best')
113     pyl.savefig(curdir+"/img/epsilon.png")
114
115 #make plots
116 #compare_methods(n=10)
117 compare_approx_n(approx_string="general") #exercise b
118 #compare_approx_n(approx_string="specific")
119 #epsilon_plots()
120 pyl.show()

```