Project 1: numerically solving a differential equation through a linear system

Daniele Brugnara <daniebru@mail.uio.no>
Martina Cammilli <martinacam@mail.uio.no>
Matteo Seclì <mattes@mail.uio.no>

September 8, 2014

Abstract

In this project we aim to solve a special kind of differential equation using a numerical procedure that allows us to express the equation through a linear system. We will study some algorithms to solve such a problem, focusing on the efficiency of the program, setting our goal more on speed than generality.

---

1   Introduction

The differential equation we're interested in studying is of the type

$$u''(x) = -f(x) \tag{1}$$

In our case we will limit our solutions using the contour conditions of $u(0) = 0$ and $u(L) = 0$, where $[0, L]$ is our domain of integration. Using Taylor expansion it is possible to express the second derivative of a function $u(x)$ as

$$u''(x) = \frac{u(x - h) - 2u(x) + u(x + h)}{h^2} + \varnothing(h^2) \tag{2}$$

We are therefore able to discretize equaition (1) using N points, obtaining:

$$u''_i = \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} = -f_i \qquad i \in \{1 \cdots N\}$$

Using the matrix representation, we can write equation (1) as

$$\begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & -1 & 2 & -1 \\ 0 & 0 & \cdots & \cdots & -1 & 2 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = h^2 \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{pmatrix} \tag{3}$$

Note how, with this system it is already implied that $f(0) = 0$ e $f(L) = 0$, since the first and last equations state that

$$h^2 f_0 = 2u_0 - u_1 = -1u_{-1} + 2u_0 - u_1$$

$$h^2 f_{N-1} = -u_{N-2} + 2u_{N-1} = -u_{N-2} + 2u_{N-1} - u_N$$

Since the boundary conditions of the differential equations state that $u_{-1} = u(0) = 0$ and $u_N = u(L) = 0$.

This linear system is indeed very particular and has a clear pattern. We will first focus on finding a solving algorithm for a general tridiagonal matrix and after we will try to implement another program to solve this particular system with the intent of lowering the number of calculation and therefore the computation time.

2 General algorithm for solving a tridiagonal matrix through back and forward substitution

A general tridiagonal system can be expressed as

$$
\begin{pmatrix}
b_0 & c_0 & 0 & 0 & \cdots & 0 \\
a_1 & b_1 & c_1 & 0 & \cdots & 0 \\
0 & a_2 & b_2 & c_2 & \cdots & 0 \\
\vdots & \vdots & & \ddots & & \vdots \\
0 & 0 & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\
0 & 0 & \cdots & \cdots & a_{N-1} & b_{N-1}
\end{pmatrix}
\begin{pmatrix}
u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1}
\end{pmatrix}
= h^2
\begin{pmatrix}
f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-2} \\ f_{N-1}
\end{pmatrix}
\tag{4}
$$

We will describe the algorithm we used for this system first for a 3×3 tridiagonal matrix, and after we will demonstrate its validity for a square tridiagonal matrix of optional dimension.

- 3×3 matrix:

$$
\left(\begin{array}{ccc|c}
b & c & 0 & f_0 \\
a & b & c & f_1 \\
0 & a & b & f_2
\end{array}\right)
\xrightarrow{\text{Passage 1}}
\left(\begin{array}{ccc|c}
1 & c/b & 0 & f_0/b \\
a & b & c & f_1 \\
0 & a & b & f_2
\end{array}\right)
$$

$$
\xrightarrow{\text{Passage 2}}
\left(\begin{array}{ccc|c}
1 & \frac{c}{b} & 0 & \frac{f_0}{b} \\
0 & \frac{b-(c/b)a}{b-(c/b)a} & \frac{c}{(b-(c/b)a} & \frac{f_1-af_0}{b-(c/b)a} \\
0 & 0 & \frac{b-\frac{ac}{b-ac/b}}{b-\frac{ac}{b-ac/b}} & \frac{f_2-af_1}{b-\frac{ac}{b-ac/b}}
\end{array}\right)
=
\left(\begin{array}{ccc|c}
1 & c/b & 0 & \frac{f_0}{b} \\
0 & 1 & \frac{c}{b-ac/b} & \frac{f_1-af_0}{b-(c/b)a} \\
0 & 0 & 1 & \frac{f_2-af_1}{b-\frac{ac}{b-ac/b}}
\end{array}\right)
$$

$$
\xrightarrow{\text{Passage 3}}
\left(\begin{array}{ccc|c}
1 & 0 & 0 & f_0/b - f_1(c/b) \\
0 & 1 & 0 & \frac{f_1-af_0}{b-(c/b)a} - f_2\frac{c}{b-ac/b} \\
0 & 0 & 1 & \frac{f_2-af_1}{b-\frac{ac}{b-ac/b}}
\end{array}\right)
$$

Now it's very simple to solve the system.

- $(n+1) \times (n+1)$ matrix.

  Before starting to demonstrate that the above passages can be done also for a $(n+1) \times (n+1)$ matrix, supposed that they work for a $n \times n$ one, we can notice that, in general, for a square matrix of optional dimension N, doing the passage 2 until the penultimate row we obtain:

$$
A_{N-1,N} = \frac{c}{bet(N-1)}
$$

  where

$$
bet(n) = b_0 - \cfrac{ac}{b_1 - \cfrac{ac}{b_2 - \cfrac{ac}{\cdots \cfrac{}{b_{n-1} - \cfrac{ac}{b_n}}}}}
$$

  (here all the $b_i's$ have the same value; the index i helps only to count them).

  Now we do the passage 2 until the last row (we focus only on the tridiagonal matrix; if we manage to obtain the unitary matrix the system is solved); we obtain:

$$
\begin{pmatrix}
1 & c/b & 0 & 0 & \cdots \\
0 & 1 & \frac{c}{b-ac/b} & 0 & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & 1 & c/bet(n) \\
\cdots & \cdots & \cdots & 0 & 1
\end{pmatrix}
\tag{5}
$$

  and simply subtracting, from the n-row, the (n+1)-row multiplied for bet(n)/c:

$$
\begin{pmatrix}
1 & c/b & 0 & 0 & \cdots \\
0 & 1 & \frac{c}{b-ac/b} & 0 & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & 1 & 0 \\
\cdots & \cdots & \cdots & 0 & 1
\end{pmatrix}
\tag{6}
$$

Now, ignoring the (n+1)-row and the (n+1)-column, we have a nxn matrix which we can bring back to the identity going on with the passage 3.

- Algorithm in C++

  Translating the above algorithm in C++ language and working only on the vector f and on the solution vector u, we obtain the following code:

```
    u[0]=f[0]/(bet=b);                                              1
for(int j = 1; j < N; j++) {                                        2
    gam[j]=c/bet;                                                   3
    bet=b-a*gam[j];                                                 4
    u[j]=(f[j]-a*u[j-1])/bet;                                       5
}                                                                   6
for (int j = (N-2); j >= 0; j--) u[j] -= gam[j+1]*u[j+1];           7
```

2.1   Particular algorithm

Using the regular Gaussian elimination algorithm we proceed to find a specific solution of our system as follows:

$$
\left(\begin{array}{cccccc|c}
2 & -1 & 0 & 0 & \cdots & 0 & f_0 \\
-1 & 2 & -1 & 0 & \cdots & 0 & f_1 \\
0 & -1 & 2 & -1 & \cdots & 0 & f_2 \\
\vdots & \vdots & & \ddots & & \vdots & \vdots \\
0 & 0 & \cdots & -1 & 2 & -1 & f_{N-2} \\
0 & 0 & \cdots & \cdots & -1 & 2 & f_{N-1}
\end{array}\right)
\longrightarrow
\left(\begin{array}{cccccc|c}
2 & -1 & 0 & 0 & \cdots & 0 & f_0 \\
0 & 3 & -2 & 0 & \cdots & 0 & 2f_1 + f_0 \\
0 & 0 & 4 & -3 & \cdots & 0 & 3f_2 + 2f_1 + f_0 \\
\vdots & \vdots & & \ddots & & \vdots & \vdots \\
0 & 0 & \cdots & 0 & N & -(N-1) & (N-1)f_{N-2} + \sum_{j=0}^{N-3}(j+1)f_j \\
0 & 0 & \cdots & \cdots & 0 & N+1 & \sum_{j=0}^{N-1}(j+1)f_j
\end{array}\right)
$$

$$(7)$$

We therefore know ahead of times the explicit form of the matrix in upper triangular form and are able to compute all the constant terms of the system as follows

$$\tilde{f}_i = \sum_{j=0}^{i}(j+1)f_j$$

However, we don't need to compute the sum every time, since we can compute $\tilde{f}_i$ knowing $\tilde{f}_{i-1}$:

$$\tilde{f}_i = (i+1)f_i + \tilde{f}_{i-1}$$

Once this forward computations are completed, it is possible to proceed with a back substitution, knowing that

$$u_{N-1} = \frac{1}{N+1}\tilde{f}_{N-1}$$

we are able to find the vector of solutions u_i

$$u_i = \frac{1}{i+2}(\tilde{f}[i] + (i+1)u[i+1])$$

Translating the algorithm in C++ code we obtain the following cycle:

```
for(int j = 1; j < N; j++) f[j]=(j+1)*f[j]+f[j-1];                             1
u[N-1] = f[N-1]/(N+1);                                                         2
int prev_idx = N;                                                             3
for(int j = N - 1; j > 0; j--) {u[j-1]=(f[j-1]+j*u[j])/prev_idx; prev_idx = j;}  4
```

Note that in the code provided here the indexes are shifted in order to reach a theoretical number of FLOPS equal to $6N$. Further we will discuss better about FLOPS in our program.

3 Solution of the differential equation

Given our differential equation

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

it is easy to solve by just integrating the function $f(x)$ and using the boundary condition to determine the value of the two integration constants. Our function is $f(x) = 100e^{-10x}$

$$u'(x) = \int -100e^{-10x}dx = 10e^{-10x} + C_1 \qquad u(x) = \int u'(x)dx = -e^{-10x} + C_1x + C_2$$

and our boundary conditions are $u(0) = -1 + C_2 = 0$ and $u(1) = -e^{-10} + C_1 + C_2 = 0$. Solving the system we obtain $C_1 = e^{-10} - 1$, $C_2 = 1$, which substituted in our family of solutions yields to our analytical solution

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x}$$

This function will be used to calculate the errors caused by our numerical approximation. In Figure 1 we can see the graph of this function as computed by solving the differential equation. Not much can be said except that the boundary conditions are satisfied.
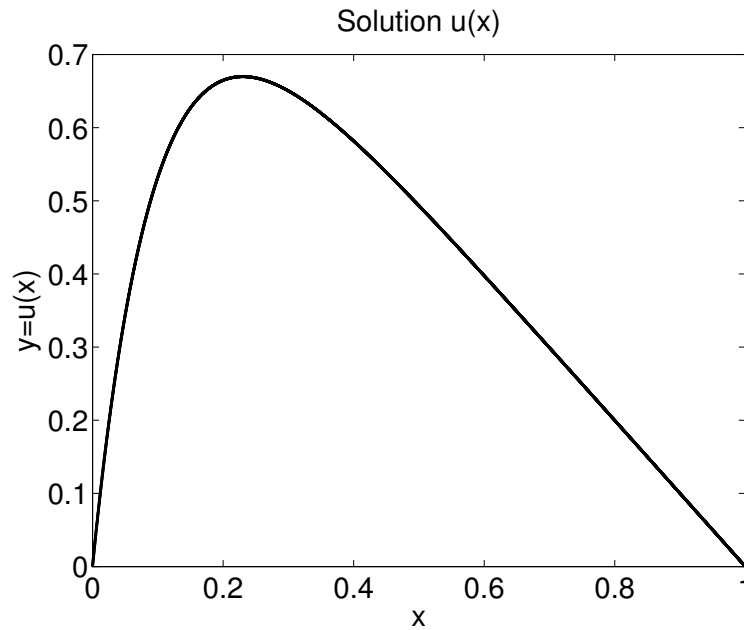
Figure 1: Graph of the solution of the differential equation, computed with 10^4 points, using the special algorithm.

4 Time

We will now study how the program performs as a function of the dimension of the matrix.
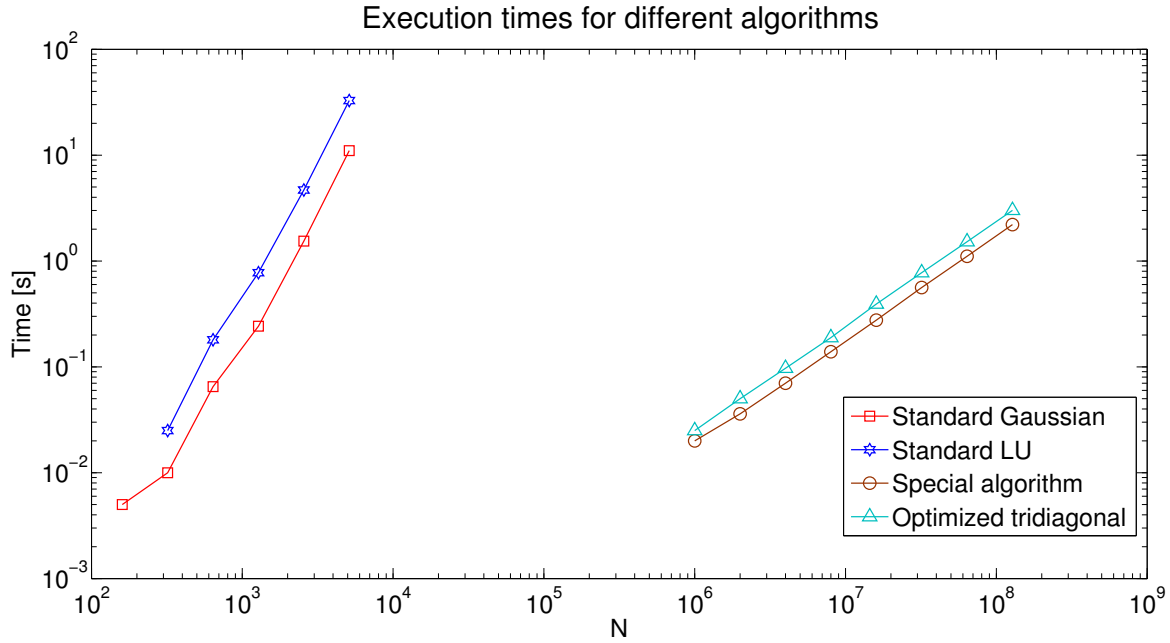
Figure 2: Elapsed time during calculation for the various algorithms on a logarithmic grid.

In Figure 2 we can notice how a specialized algorithm is indeed able to significantly cut down the time required for the computation of the solution. In fact, we can easily notice how, with a standard Gaussian elimination, the required time for solving a 2×10^2 dimensional matrix is the same it takes for the specialized algorithm to compute 10^6 points. This yields clearly to a much higher degree of precision and a better deployment of resources. As a matter of fact, counting the number of operations contained in the specialised algorithm is $8N$, whereas for the LU decomposition it should amount to $N^2 + N^3$.

5   Errors

It is now interesting to analyse the error that we do in numerical approximation as a function of the number of points. It is intuitive that, as much as we restrict the step length h (that is the same thing as increasing the number of points) the error gets smaller and smaller. This behaviour is shown if Figure 3, where we plotted in a log-log scale the maximum percentage error calculated as

$$\epsilon_i = \left| \frac{v[i]}{u[i]} - 1 \right|$$

($v[i]$ is the numerical solution, $u[i]$ is the analytical one).
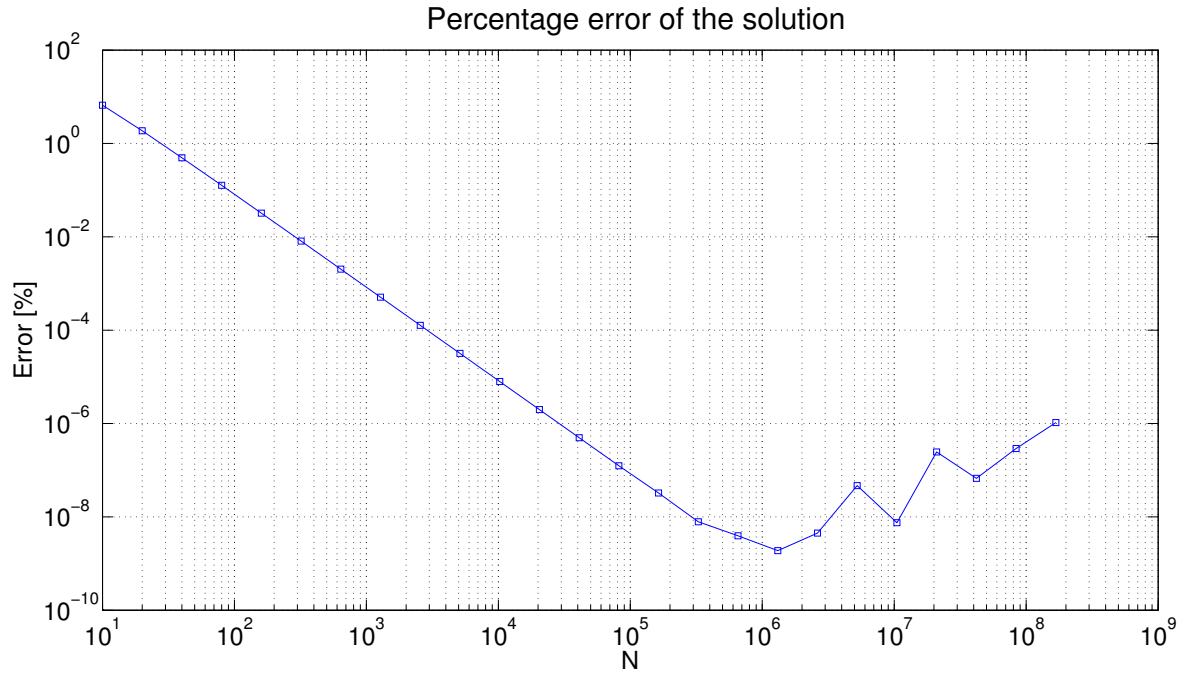
Figure 3: Maximum percentage error of the numerical solution as a function of the number of points.

You see that the error lowers until $N = 10^5$, then increases again. This a typical example of *loss of precision*; as our numerical solution gets closer to the analytical one, the ratio $v[i]/u[i]$ gets closer to one, and as a result in calculating ϵ_i we perform a subtraction between two almost identical (in our choice of precision) numbers. This causes a loss in terms of significant digits that explains the behaviour of the plot for big N. It is also evident that we get the lowest significant relative error for $N \simeq 10^6$. Relative errors for $N > 10^6$ are not worth trusting, due to the loss of precision explained above.

6 C++ Code

```cpp
#include <iostream>                                                           1
#include <fstream>                                                            2
#include <armadillo>                                                          3
#include <cstdlib>                                                            4
#include <cmath>                                                              5
#include <ctime>                                                              6
using namespace std;                                                          7
using namespace arma;                                                         8
namespace use {int onealg = 0; int out = 1;}                                  9
// 'fill_matrix' fills the matrix A in a tridiagonal form.                   10
void fill_matrix(mat& A, int N) {                                            11
//Fill the matrix                                                            12
A(0,0) = 2;                                                                  13
A(0,1) = -1;                                                                 14
for(int i = 1; i < N-1; i++){                                                15
A(i,i-1) = -1;                                                               16
A(i,i) = 2;                                                                  17
A(i,i+1) = -1;                                                               18
}                                                                            19
A(N-1,N-2) = -1;                                                             20
A(N-1,N-1) = 2;                                                              21
}                                                                            22
// 'solve_gaus' solves the system with a standard gaussian decomposition    23
void solve_gaus(vec& u, vec& f, int N){                                      24
//Start timing                                                              25
clock_t t;                                                                   26
t = clock();                                                                 27
//Define our matrix and initialize it                                       28
mat A(N,N);                                                                  29
A.zeros();                                                                   30
fill_matrix(A,N);                                                            31
//Just the decomposition                                                    32
u = solve(A,f);                                                              33
A.reset();                                                                   34
//Stop timing                                                               35
t = clock() - t;                                                            36
cout <<"Elapsed time (solve_gaus):\t\t" <<((float)t)/CLOCKS_PER_SEC <<"s." <<endl;   37
}                                                                            38
// 'solve_lu' solves the system with a standard LU decomposition            39
void solve_lu(vec& u, vec& f, int N){                                        40
//Start timing                                                              41
clock_t t;                                                                   42
t = clock();                                                                 43
//Define our matrix and initialize it                                       44
mat A(N,N);                                                                  45
A.zeros();                                                                   46
fill_matrix(A,N);                                                            47
//Define workspace matrices                                                 48
mat L(N,N), U(N,N), P(N,N);                                                  49
//Do the decomposition                                                      50
lu(L, U, P, A);                                                              51
A.reset();                                                                   52
//Just solve the system                                                     53
vec b;                                                                       54
b = solve(L,P*f);                                                            55
L.reset();                                                                   56
P.reset();                                                                   57
u = solve(U,b);                                                              58
U.reset();                                                                   59
//Stop timing                                                               60
t = clock() - t;                                                            61
cout <<"Elapsed time (solve_lu):\t\t" <<((float)t)/CLOCKS_PER_SEC <<"s." <<endl;   62
}                                                                            63
// 'solvetrid' is a function that solves a linear sistem in 'u' relative to a   64
// tridiagonal matrix with diagonal elements equal to 'b', subdiagonal elements   65
// equal to 'a' and superdiagonal elements equal to 'c'. Returns the result   66
// in the vector 'u', overwriting its elements. The algorithm performs ~8N FLOPS.   67
void solvetrid(int& N, float& a, float& b, float& c, vec& u, vec& f){        68
// Start timing                                                             69
clock_t t;                                                                   70
t = clock();                                                                 71
// Define the variable 'bet', that is just the denominator of 'gam',        72
```

```cpp
// and 'gam' itself, that is a workspace vector                              73
double bet;                                                                 74
vec gam(N);                                                                 75
// Start forward substitution                                              76
u[0]=f[0]/(bet=b);                                                         77
for(int j = 1; j < N; j++) {                                              78
gam[j]=c/bet;                                                              79
bet=b-a*gam[j];                                                            80
u[j]=(f[j]-a*u[j-1])/bet;                                                  81
}                                                                          82
// Just one-line backward substitution                                    83
for (int j = (N-2); j >= 0; j--) u[j] -= gam[j+1]*u[j+1];                  84
// Stop timing and print elapsed time                                     85
t = clock() - t;                                                          86
cout << "Elapsed time (solvetrid):\t\t" << ((float)t)/CLOCKS_PER_SEC << "s." << endl;  87
// Free space                                                             88
gam.reset();                                                              89
}                                                                          90
// 'solve_special' is a function that solves a special linear system      91
// relative to a tridiagonal matrix with b = 2 and a = c = -1. The        92
// solution has been found analytically, and once the pattern in          93
// the solution was recognized, it has been coded here. Warning! It       94
// overwrites 'u' and 'f', so make a copy before calling the function     95
// if you want to re-use them. The alogrithm performs ~6N FLOPS.          96
void solve_special(int& N, vec& u, vec& f){                               97
// Start timing                                                          98
clock_t t;                                                               99
t = clock();                                                            100
for(int j = 1; j < N; j++) f[j]=(j+1)*f[j]+f[j-1];                      101
u[N-1] = f[N-1]/(N+1);                                                  102
int prev_idx = N;                                                       103
for(int j = N - 1; j > 0; j--) {u[j-1]=(f[j-1]+j*u[j])/prev_idx; prev_idx = j;}  104
// Stop timing and print elapsed time                                  105
t = clock() - t;                                                       106
cout << "Elapsed time (solve_special):\t\t" << ((float)t)/CLOCKS_PER_SEC << "s." << endl;  107
}                                                                        108
// 'split' is a function that discretizes the function 'func',         109
// storing its values in N points from 0 to 1 in the vector 'f'.       110
// Grid points are stored in the vector 'x'.                           111
void split(vec& f, vec& x, int& N) {                                   112
// Define points spacing and calculate the grid points                 113
// and the value of func in those points                               114
double h = 1.0/(N+1);                                                   115
double h_square = pow(h,2);                                            116
for(int i = 0; i < N; i++){                                            117
x[i] = (i+1)*h;                                                        118
f[i] = h_square*100*exp(-10*x[i]);                                     119
}                                                                       120
}                                                                       121
// 'relative_error' calculates the relative error with respect to the  122
// theoretical value 'u_th(x)'.                                        123
vec relative_error(vec& u, vec& x, int& N) {                          124
vec err(N);                                                           125
err[0] = 0;                                                           126
for(int i = 0; i <= N-1; i++){                                       127
err[i] = abs( u[i]/(1-(1-exp(-10))*x[i]-exp(-10*x[i])) - 1 );        128
}                                                                     129
return err;                                                          130
}                                                                     131
// 'main' takes as first argumt the number of points that the program 132
// will use during the calculation. Use 'onealg 1' as second argument 133
// if you want to use only one algorithm and write to the output file. 134
// Use 'anealg 0' if you want to use only one algorithm and don't write 135
// to the output file.                                                136
int main(int argc, char *argv[])                                     137
{                                                                     138
int N = atoi(argv[1]);                                               139
// Perform some checks in the optional argument.                      140
if(argc == 4) {                                                       141
use::out = atoi(argv[3]);                                            142
if(strcmp(argv[2], "onealg") == 0 && (strcmp(argv[3], "1") == 0||strcmp(argv[3], "0") == 0)) use::  143
    onealg = 1;
else cout << "Wrong optional argument given. Use 'onealg 1' if you want to use only one algorithm (  144
    the fastest) and write to file; use 'onealg 0' if you instead want to write to the output file."
     << endl;
```

```
}                                                                              145
// Define the elements of the matrix related to the differential equation      146
float a = -1.0;                                                                147
float b = 2.0;                                                                 148
float c = -1.0;                                                                149
// Initialize the solution vector 'u' with zeros and the vector 'f'            150
// of the function values                                                      151
vec u = zeros<vec>(N);                                                         152
vec f(N), x(N);                                                               153
//for(int i = 0; i < N; i++) f[i] = i+1;                                       154
// Discretize and define workspace vectors                                     155
split(f, x, N);                                                                156
vec u_temp(N), f_temp(N), err(N);                                             157
// Compare algorithms only if we want to do benchmarks.                        158
// This is to save memory if we want just to have grid numbers.                159
if(use::onealg == 0) {                                                         160
// Solve using 'solve_lu' and 'solve_gaus'                                     161
if (N <= 10000) {                                                             162
solve_lu(u,f,N);                                                               163
solve_gaus(u,f,N);                                                            164
}                                                                              165
// Solve using 'tridig'                                                        166
solvetrid(N, a, b, c, u, f);                                                   167
}                                                                              168
// Solve using 'solve_special'                                                 169
solve_special(N, u, f);                                                        170
f.reset();                                                                     171
// Compute relative error only if 'onealg' is enabled, to speed up benchmarks  172
if(use::onealg == 1) {                                                         173
err = relative_error(u, x, N);                                                 174
cout << "Maximum relative error: " << err.max()*100 << "%" << endl;            175
}                                                                              176
// Write the resulting points on the output file                               177
if(use::onealg == 1 && use::out == 1) {                                        178
// Write the 'x' grid-points to the output file                                179
ofstream X;                                                                    180
X.open("X.txt");                                                              181
X << x;                                                                        182
X.close();                                                                     183
// Write the 'u' grid-points to the output file                                184
ofstream U;                                                                    185
U.open("U.txt");                                                              186
U << u;                                                                        187
U.close();                                                                     188
// Write the error bars to the output file                                     189
ofstream E;                                                                    190
E.open("E.txt");                                                              191
E << err;                                                                      192
E.close();                                                                     193
}                                                                              194
return 0;                                                                      195
}                                                                              196
```