TANA21 - Scientific Computing

Programming Project Version Zwei

Professor: Andrew Ross-Winters

Group Leader: Fredrik Laurén

Authors:
Nikolas Tsigkas
Rinor Zylfijaj

Student ID:
nikts531
rinzy571

March 6, 2021

Contents

1 Problem Statement

The aim of this project report is to implement and evaluate a numerical method of solving an ordinary differential equation (ODE) as a system of linear equations. The implementation will be done in MATLAB (all relevant parts of code used in the project are included in the appendices).

2 Mathematical Description of the Problem

The ODE that we seek to solve numerically is of second order and has the form shown below. Differential equations of this kind are applicable in multiple sciences, for instance governing the behaviour of beams in solid mechanics.

$$\begin{aligned} &y''(x) = p(x)y'(x) + q(x)y(x) + r(x) \\ &x \in (a, b) \\ &y(a) = \alpha \\ &y(b) = \beta \end{aligned}$$

The first and second derivatives of $y(x)$, evaluated at a point x can be approximated using a finite difference approximation. Using the central difference approximation, we obtain:

$$\begin{aligned} y'(x) &\approx \frac{y(x+h) - y(x-h)}{2h} \\ y''(x) &\approx \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} \end{aligned}$$

This approximation will have an error of $\mathcal{O}(h^2)$, which can be derived from taking the Taylor series expansion of $y(x+h)$ and $y(x-h)$. By breaking down the interval (a, b) given in the problem into n sub intervals defined by $n+1$ points $(x_0, x_1, ..., x_{n-1}, x_n)$ with equal spacing between two adjacent points, we have $h = \frac{b-a}{n}$ and we can evaluate the derivatives at the interior points as following:

$$y_i' \approx \frac{y_{i+1} - y_{i-1}}{2h} \quad y_i'' \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

Where we have also introduced the shortened notation $y_i = y(x_i)$. Inserting these expressions into the general form of the ODE and rearranging the terms results in the expression below, where the same, shortened notation is used.

$$(1 + \frac{p_i h}{2})y_{i-1} - (2 + q_i h^2)y_i + (1 - \frac{p_i h}{2})y_{i+1} = r_i h^2 \quad \forall i = 1, .., n-1$$

However, when $i = 1$ or $n-1$ one of the terms in the left hand side is already known as it will be a boundary value. Moving that term, so that the left hand side consists only of unknowns, for these two cases results in:

$$\begin{aligned} &-(2 + q_1 h^2)y_1 + (1 - \frac{p_1 h}{2})y_2 = r_1 h^2 - (1 + \frac{p_1 h}{2})\alpha \\ &(1 + \frac{p_{n-1} h}{2})y_{n-2} - (2 + q_{n-1} h^2)y_{n-1} = r_{n-1} h^2 - (1 - \frac{p_{n-1} h}{2})\beta \end{aligned}$$

The resulting system of equations can now be represented compactly in matrix-vector form as $\mathbf{Ay} = \mathbf{b}$ where $\mathbf{A}$, $\mathbf{y}$, $\mathbf{b}$ are as follows:

$$\mathbf{A} = \begin{pmatrix} -(2+q_1h^2) & (1-\frac{p_1h}{2}) & 0 & \cdots & \cdots & 0 \\ (1+\frac{p_2h}{2}) & -(2+q_2h^2) & (1+\frac{p_2h}{2}) & 0 & \cdots & 0 \\ 0 & (1+\frac{p_3h}{2}) & -(2+q_3h^2) & (1-\frac{p_3h}{2}) & \cdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \cdots & \cdots & (1+\frac{p_{n-2}h}{2}) & -(2+q_{n-2}h^2) & (1-\frac{p_{n-2}h}{2}) \\ 0 & \cdots & \cdots & 0 & (1+\frac{p_{n-1}h}{2}) & -(2+q_{n-1}h^2) \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_{n-1} \end{pmatrix} \quad \mathbf{b} = h^2 \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ \vdots \\ r_{n-1} \end{pmatrix} - \begin{pmatrix} (1+\frac{p_1h}{2})\alpha \\ 0 \\ \vdots \\ 0 \\ (1-\frac{p_{n-1}h}{2})\beta \end{pmatrix}$$

3 Description of the Algorithm

3.1 Creating A and b

The matrix $\mathbf{A}$ and vector $\mathbf{b}$ are generated by the function `MVrep.m` (A.1), which takes as input the functions $p(x)$, $q(x)$, $r(x)$ as anonymous functions, the boundary values, bounds and number of sub-intervals as scalar values. The function begins with finding h and creating $n+1$ uniformly spaced nodes, for which then, p, q and r are evaluated. Finding the vector $\mathbf{b}$ (denoted `rhs` in the code) is then straight forward as h^2 multiplied by a vector containing r evaluated at all interior points, minus the extra terms at the first and last entry of the vector arising due to the boundaries.

It can be noticed that A is tridiagonal. To build the matrix, we can thus construct one diagonal at a time and add them together using the `diag` function three times. We do this using the fact that:

$$\begin{aligned} a_{i,i} &= -(2+q_ih^2) \quad i = 1,..,n-1 \\ a_{i,i-1} &= (1+\frac{p_ih}{2}) \quad i = 2,..,n-1 \\ a_{i,i+1} &= (1-\frac{p_ih}{2}) \quad i = 1,..,n-2 \end{aligned}$$

The MATLAB function also returns all the nodes `xVals` as these will be used to find the function values for the true solution, as well as for plotting the results. Finally, when working with the code, one must pay extra attention to the indexing, as the mathematical notation we are working with starts indexing at zero while vectors and matrices in MATLAB always start at one.

3.2 Solving Ay=b

The matrix $\mathbf{A}$ is a square matrix, meaning we can, as long as we are working with a non-singular matrix, find a solution numerically through for instance LU-factorization where we decompose A into an upper and a lower triangular matrix. But we have also noticed that it is tridiagonal. Taking this into account, we can modify the standard LU algorithm. This modification will result in the Thomas algorithm, which is performed by the function `thomas.m` (A.2). The output matrices $\mathbf{L}$ and $\mathbf{U}$ are created iteratively from A as follows, for $i = 1,..,n-1$:

$$\begin{aligned} u_{1,1} &= a_{1,1} \\ l_{i+1,i} &= \frac{a_{i+1,i}}{u_{i,i}} \\ u_{i+1,i+1} &= a_{i+1,i+1} - l_{i+1,i}a_{i,i+1} \end{aligned}$$

The final appearance of $\mathbf{L}$ and $\mathbf{U}$ will then be:

$$L = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & 0 \\ l_{2,1} & 1 & 0 & \cdots & \cdots & 0 \\ 0 & l_{3,2} & 1 & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & l_{n,n-1} & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_{1,1} & a_{1,2} & 0 & \cdots & 0 \\ 0 & u_{2,2} & a_{2,3} & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & u_{n-1,n-1} & a_{n-1,n} \\ 0 & \cdots & \cdots & 0 & u_{n,n} \end{pmatrix}$$

When implementing this on a machine, we do not need to store and update $\mathbf{L}$ and $\mathbf{U}$ during the entire process. Instead, we can efficiently handle all the values of both matrices inside one array. The array starts of as the matrix A and for each step, the value to the left of the diagonal element (i+1,i) and the diagonal element itself (i+1,i+1) will be replaced by $l_{i+1,i}$ and $u_{i+1,i+1}$ respectively. When the iterations are finished, we obtain $\mathbf{L}$ and $\mathbf{U}$ by "splitting" the array below the main diagonal with the `tril` and `triu` functions, and adding $\mathbf{I}_n$, the n dimensional identity matrix, to $\mathbf{L}$. Here, the index notation is consistent between the mathematical and MATLAB representation.

The matrices $\mathbf{L}$ and $\mathbf{U}$ now have the property that $\mathbf{LU} = \mathbf{A}$. Solving the problem $\mathbf{Ay} = \mathbf{b}$ can now be replaced by solving the following two problems:

$$\mathbf{Lc} = \mathbf{b} \quad \mathbf{Uy} = \mathbf{c}$$

Because of the shape of these matrices, solving these two systems of equations is easily done through first forward and then backwards substitution to find $\mathbf{c}$ and $\mathbf{y}$ respectively. This is done by the functions `forwardsub.m` (A.3) and `backsub.m` (A.4), via the following iterations:

$$\begin{aligned} c_1 &= b_1 \\ c_i &= b_i - l_{i,i-1}c_{i-1} \qquad i = 2, .., n \\ y_n &= \frac{c_n}{u_{n,n}} \\ y_i &= \frac{c_i - u_{i,i+1}y_{i+1}}{u_{i,i}} \quad i = n-1, .., 1 \end{aligned}$$

Finally, the program `main.m` (A.5) puts everything together to find the vector `y_aprVals` which is the numerical solution for the interior points and the boundary values at the end points and compares it to the vector of true values of the function (which is given in its closed form as an anonymous function) `y_exVals` by measuring error and creating plots.

4 Results

4.1 Computational cost for the Thomas algorithm

Firstly, an evaluation of the computational cost of solving a system of linear equations with the Thomas algorithm was performed in the program `thomastest.m` (A.6), which also verifies that the algorithm works. By creating arbitrary tridiagonal matrices of different sizes and measuring the time taken for the machine to solve them using `thomas.m`, `forwardsub.m` and `backsub.m`, we could study the relationship between matrix size and computational time. To reduce the error when measuring the time, the time to perform the algorithm is measured 50 times and is then averaged. Figure 1 suggests that the computational cost is $\mathcal{O}(n)$ due to the observed relatively linear dependence.

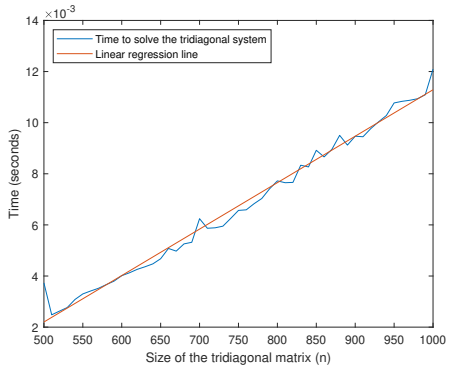

Figure 1: Computational cost of the Thomas algorithm when run on a 7th generation Intel Core I7 CPU.

4.2 Solving for a specific ODE

The algorithm is evaluated for the functions, bounds and boundary values displayed in Table 1. The true solution for this particular ODE is then the following:

$$y(x) = \frac{4}{x} - \frac{2}{x^2} + ln(x) - \frac{3}{2}$$

$p(x)$	$q(x)$	$r(x)$	a	b	α	β
$-\frac{4}{x}$	$-\frac{2}{x^2}$	$2\frac{ln(x)}{x^2}$	1	2	0.5	$ln(2)$

Table 1: Functions, bounds and boundaries of the ODE

Using the known exact function, the accuracy of the numerical approximation can be evaluated. One way is to measure the infinity norm. If $\mathbf{y}^{(ex)}$ denotes a vector containing the true solution on the set of nodes $(x_0, .., x_n)$, then the infinity norm is defined as:

$$||\mathbf{y}_{ex} - \mathbf{y}||_\infty = \max_i\{|y_i^{(ex)} - y_i|\}$$

By solving the matrix vector problem for an increasing set of sub intervals (thus decreasing the step-size h as they are inversely proportional according to the definition of h provided in Section 2), the impact that this has on the error can be examined. Plotting the infinity norm against n^{-2} yields the linear curve seen to the left in Figure 2, verifying the claim made previously that the central difference approximation is second order convergent. The solution vector $\mathbf{y}$ can also be examined graphically compared to the true solution, which is seen to the right in Figure 2.

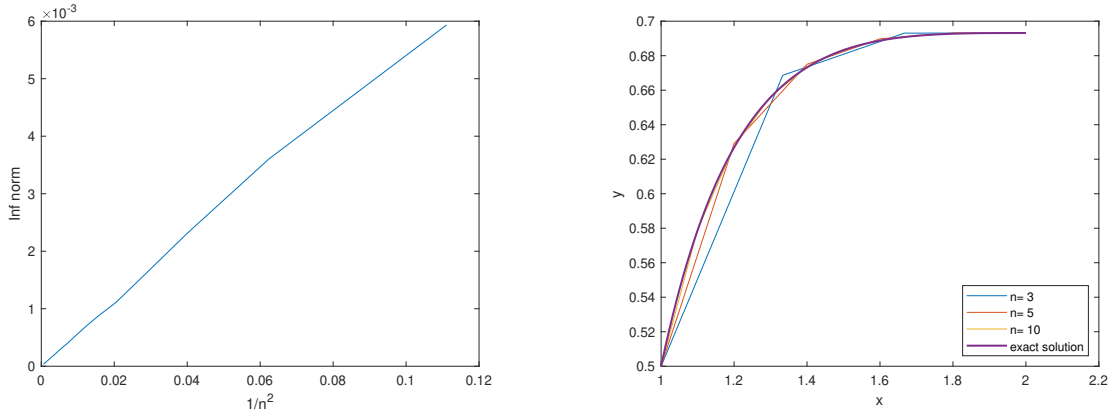

Figure 2: Error vs. n^{-2} for n between 5 and 80 (left) and $y(x)$ compared to numerical solutions (right)

The values of the true solution and our approximation when $n = 10$ are shown in Table 2

Value of x	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
Exact	0.500	0.5788	0.6268	0.6559	0.6732	0.6832	0.6888	0.6915	0.6927	0.6931	0.6931
Approximation	0.500	0.5792	0.6273	0.6564	0.6736	0.6836	0.6890	0.6917	0.6928	0.6931	0.6931

Table 2: Exact values of the function vs. numerical approximation for $n = 10$

5 Discussion

After conducting the discretization of the ODE using finite approximations such as the central difference approximation we found our results matched our expectations. To our knowledge the central difference approximation is second order accurate and we showed that we could apply it to estimate the function at all inner points with second degree accuracy. As the boundary conditions give the function values at the bounds, we did not have to use lower order approximations such as backward and forward differencing at any point, which is one of the issues with using the central difference approximation in some cases.

Furthermore, we could solve the system of n linear equations at computational cost $\mathcal{O}(n)$. Had we proceeded to solve the system with standard LU-factorization, the computational cost would have been $\mathcal{O}(n^3)$. This shows that we save a lot of time due to the fact that we know that our matrix is tridiagonal. Because of this property, we only have to do one division, one multiplication and one subtraction for each row (as described in Section 3.2) to get $\mathbf{L}$ and $\mathbf{U}$, which explains the linear dependence on n.

A MATLAB code

A.1 MVrep.m

```
function [A, xVals, rhs] = MVrep(p, q, r, y0, yn, a, b, n)

% y'' = p(x)*y' + q(x)*y + r(x) (p,q,r passed as anonymous functions)
% [a,b] is the interval, n are the number of steps
% y0 and yn are the values at the boundaries

% A: (n-1)x(n-1) matrix
%xVals: (n+1)x1 vector
% b: (n-1)x1 vector

h = (b-a)/(n);

xVals = zeros(n+1,1);
xVals(1) = a;
for i = 2:n+1
    xVals(i) = xVals(i-1) + h;
end

pVals = p(xVals);
qVals = q(xVals);
rVals = r(xVals);

rhs = zeros(n-1,1);

rhs = h^2*rVals(2:n); %h^2 * r at  interior points
rhs(1) = rhs(1) - (1+pVals(2)*h/2)*y0; %lower bound
rhs(end) = rhs(end) - (1-pVals(n)*h/2)*yn; % upper bound

main_diag = -qVals(2:n)*h^2 -2*ones(n-1,1);
lower_diag = pVals(3:n)*h/2 + ones(n-2,1);
upper_diag = -pVals(2:n-1)*h/2 + ones(n-2,1);

A = diag(main_diag,0) + diag(upper_diag,1) + diag(lower_diag,-1);
```

A.2 thomas.m

```
function [L,U] = thomas(A)
% LU = A

     n  = length(A);
   for j = 1:n-1
      A(j+1,j) = A(j+1,j)/A(j,j); %L-component
      A(j+1,j+1) = A(j+1,j+1) - A(j+1,j)*A(j,j+1); %diagonal U-component
   end

   U = triu(A);
   L = eye(n) + tril(A,-1);
end
```

A.3 forwardsub.m

```
function [c] = forwardsub(L,rhs)

n = length(L);
c = zeros(n,1);
c(1) = rhs(1);

for i = 2:n
    c(i) = rhs(i)-L(i,i-1)*c(i-1);
end
end
```

A.4 backsub.m

```
function [yVals] = backsub(U,c)

n = length(U);
yVals = zeros(n,1);
yVals(end) = c(end)/U(n,n);

for i = n-1:-1:1
    yVals(i) = (c(i)-U(i,i+1)*yVals(i+1))/U(i,i);
end
end
```

A.5 main.m

```
p = @(x) -4./x;
q = @(x) -2./x.^2;
r = @(x) 2*log(x)./(x.^2);
a = 1; %Lower & upper bound
b = 2;
y1 = 1/2; %Boundary values
y2 = log(2);
n = linspace(3,40,38); %Values of n that we solve for

y_ex_f = @(x) (4./x - 2./(x.^2) + log(x) -3/2); %Exact solution

e = zeros(length(n),1); %error, inf norm
er = zeros(length(n),1); %error ratio
for i = 1:length(n)
    [A, xVals, rhs] = MVrep(p, q, r, y1, y2, a, b, n(i));
    y_exVals = y_ex_f(xVals);
    y_aprVals = zeros(n(i)+1,1);
    y_aprVals(1) = y1;
    y_aprVals(end) = y2;
    %LU-factorize A with Thomas
    [L, U] = thomas(A);
    %Forward, backward (Lc=rhs, UyVals=c)
    c = forwardsub(L, rhs);
    y_aprVals(2:n(i)) = backsub(U,c);
    e(i) = norm(y_exVals - y_aprVals, Inf);
    plot(xVals, y_aprVals)
    hold on
    if i > 1
        er(i) = e(i-1)/e(i);
    end
end

x100 = linspace(a,b);
plot(x100, y_ex_f(x100), "Linewidth", 1.5) %Add the real function to the plot
xlabel("x");
ylabel("y");
legend("n= 3","n= 5","n= 10", "exact solution", "location", "southeast")

plot(1./n.^2,e)
xlabel("1/n^2");
ylabel("Inf norm");
```

A.6 thomastest.m

```
%Does it work
N=6
 %create arbitrary tridiagonal matrices
 main= N*ones(N,1) + rand(N,1);
 upper= rand(N-1,1);
 lower= rand(N-1,1);
 A = diag(main,0) + diag(upper,1) - diag(lower,-1);

%create a rhs which has the solution of the 1-vector
%compare to the solution from Thomas
known_x= ones(N,1);
manuf_b= A*known_x;

[L,U]=thomas(A);
c=forwardsub(L,manuf_b);
xthom=backsub(U,c)

%Cost analysis
N=500:10:1000;
ts=0*N;

for j=1:length(N)

    main= N(j)*ones(N(j),1) + rand(N(j),1);
    upper= rand(N(j)-1,1);
    lower= rand(N(j)-1,1);

    A = diag(main,0) + diag(upper,1) - diag(lower,-1);

    known_x= ones(N(j),1);
    manuf_b= A*known_x;

    %Solve 50 times and take average
    tic
    for i = 1:50

    [L,U]=thomas(A);
    c=forwardsub(L,manuf_b);
    xthom=backsub(U,c);

    end
   t=toc;
   ts(j)=t/50;
end

%regression line
linfit = polyfit(N,ts,1);
plot(N,ts,N, linfit(1)*N + linfit(2))
xlabel("Size of the tridiagonal matrix (n)")
ylabel("Time (seconds)")
legend("Time to solve the tridiagonal system", "Linear regression line", "
    location", "northwest")
```