Two-point boundary value problem

Samuel E. Andersson, David Wiman

19 October 2020

Contents

1 Problem statement

The goal of this project is to give a numerical approximation to an ordinary differential equation (ODE) using Matlab and confirming its accuracy compared to the analytically derived solution. The ODE that will be solved is the following:

$$y'' = cos(x)y' - sin(x)y, \quad 0 \le x \le \frac{3\pi}{2} \quad with \quad y(0) = 1 \quad and \quad y\left(\frac{3\pi}{2}\right) = \frac{1}{e},$$

which has the known analytical solution $y = e^{sin(x)}$.

2 Mathematical description

"Since Newton, mankind has come to realize that the laws of physics are always expressed in the language of differential equations" - Mathematician Steven Strogatz.

The two-point boundary value problem has numerous applications in all of the natural sciences, with one of the more famous ones being Lambert's problem in astrophysics, where a trajectory in the two-body problem is sought. Other applications include problems such as load and deformation of a beam over time or the heating of an electrical wire.

However, ODEs are, in general, not analytically solvable. Because of this, a method for numerically approximating them is necessary. There are numerous methods, and in this project central finite difference with biasing is used to approximate the derivatives and the *Successive Over-relaxation* method is used to solve the linear system that arises. When an approximation is reached, it must also be confirmed. In this project, the confirmation will be done using the known analytical solution.

3 Description of the algorithms

Let a and b be the lower and upper bounds respectively where a solution for the ODE is sought, and let n be the number of nodes. h will henceforth be called the time step.
To approximate the solution to the ODE, the different derivatives for inner points must first be approximated using central finite difference:

$$y_i' = \frac{1}{2h}(y_{i+1} - y_{i-1}) \quad and \quad y_i'' = \frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}),$$

where

$$x_i = a + ih \quad and \quad y_i = y(x_i) \quad , \quad i = 1, 2, ..., n-2, n-1 \quad , \quad h = \frac{b-a}{n}.$$

The approximation can then be substituted into the original equation forming:

$$\frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) = cos(x_i)\frac{1}{2h}(y_{i+1} - y_{i-1}) - sin(x_i)y_i.$$

Moving terms around and factoring out the function evaluations:

$$y_{i+1}\left(\frac{1}{h^2} - \frac{cos(x_i)}{2h}\right) + y_i\left(\frac{-2}{h^2} + sin(x_i)\right) + y_{i-1}\left(\frac{1}{h^2} + \frac{cos(x_i)}{2h}\right) = 0$$

This can then be written as a linear system of equations on the form:

$$\mathbf{Ay} = \mathbf{b}$$

where **A** is a square, diagonally dominant, tridiagonal matrix, **y** is the column vector of function values and **b** is a right hand side vector. The first and last rows of **A** only contains ones and zeros since the end point values are known. This can also be seen in the right hand side vector **b** having none-zero elements in its first and last row.

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & ... & .... & 0 \\ \frac{1}{h^2} + \frac{cos(x_1)}{2h} & \frac{-2}{h^2} + sin(x_1) & \frac{1}{h^2} - \frac{cos(x_1)}{2h} & 0 & \vdots \\ 0 & \frac{1}{h^2} + \frac{cos(x_2)}{2h} & \frac{-2}{h^2} + sin(x_2) & \frac{1}{h^2} - \frac{cos(x_2)}{2h} & \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \ddots \\ & 0 & \frac{1}{h^2} + \frac{cos(x_{n-1})}{2h} & \frac{-2}{h^2} + sin(x_{n-1}) & \frac{1}{h^2} - \frac{cos(x_{n-1})}{2h} \\ 0 & ... & ... & 0 & 1 \end{pmatrix},$$

$$\mathbf{y} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ ... \\ ... \\ y_{n-1} \\ y_n \end{pmatrix} \quad and \quad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ ... \\ ... \\ 0 \\ e^{-1} \end{pmatrix}.$$

These matrices are created in the ProjectODE.m file.

To solve a linear system $\mathbf{Ay} = \mathbf{b}$ where the matrix **A** is diagonally dominant, the *Successive Over-relaxation (SOR)* method will be used. The SOR method is an iterative Gauss-Seidel method which takes the form

$$y_i^{(k+1)} = (1-\omega)y_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}y_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}y_j^{(k)}\right),$$

where $\omega \in [1, 2]$ is the relaxation factor, with the optimal value for ω depending on the matrix. If $\omega = 1$ then the SOR-method turns into ordinary Gauss-Seidel and if $\omega = 2$ then the SOR-method will never converge. The stopping condition for the iteration is

$$\|\mathbf{b} - \mathbf{Ay^{(k)}}\|_2 \leq tol\|\mathbf{b}\|_2 \quad where \quad tol = 10^{-11}.$$

This is implemented in the SuccessiveOverRelaxation.m file which is called upon in ProjectODE.m to solve the system and return the approximated function values.

4 Results

With the algorithm described in chapter 3 implemented in Matlab R2020a, the approximate solutions to the ODE were computed with a varying number of nodes. Here, the approximate solution is compared to the known analytical solution to the ODE.

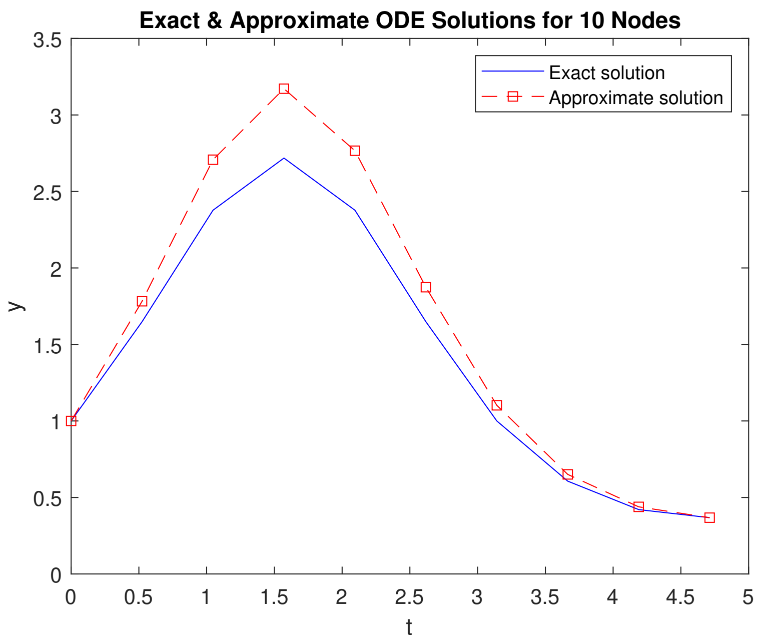

Figure 1: SOR-approximate and exact solution for 10 nodes.

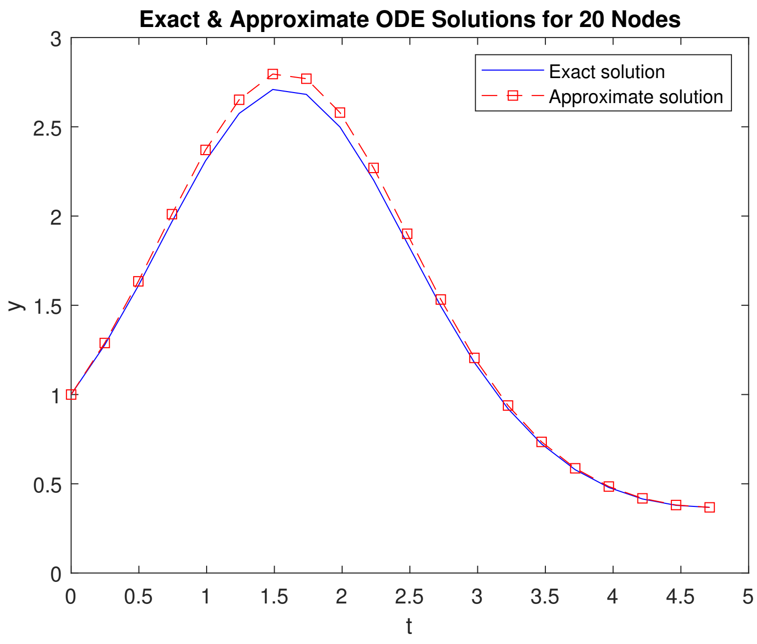

Figure 2: SOR-approximate and exact solution for 20 nodes.

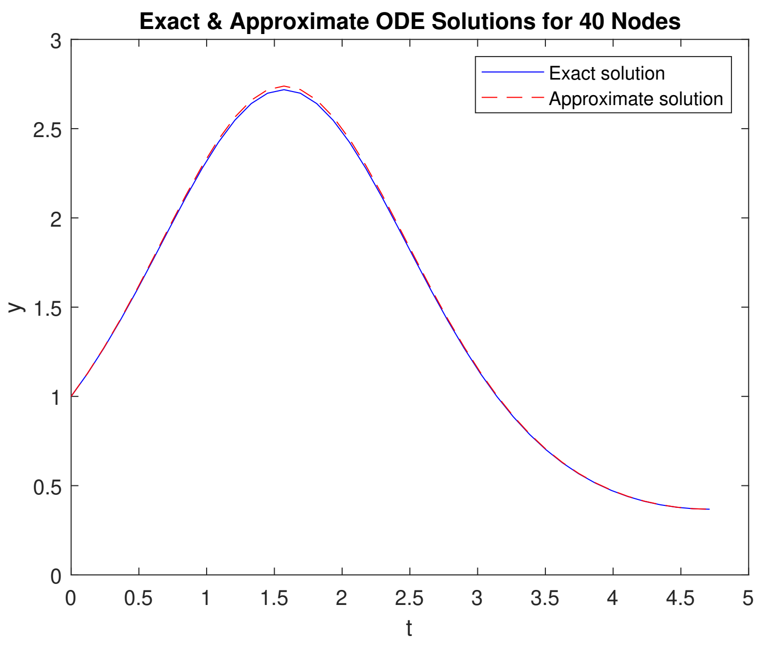

Figure 3: SOR-approximate and exact solution for 40 nodes.

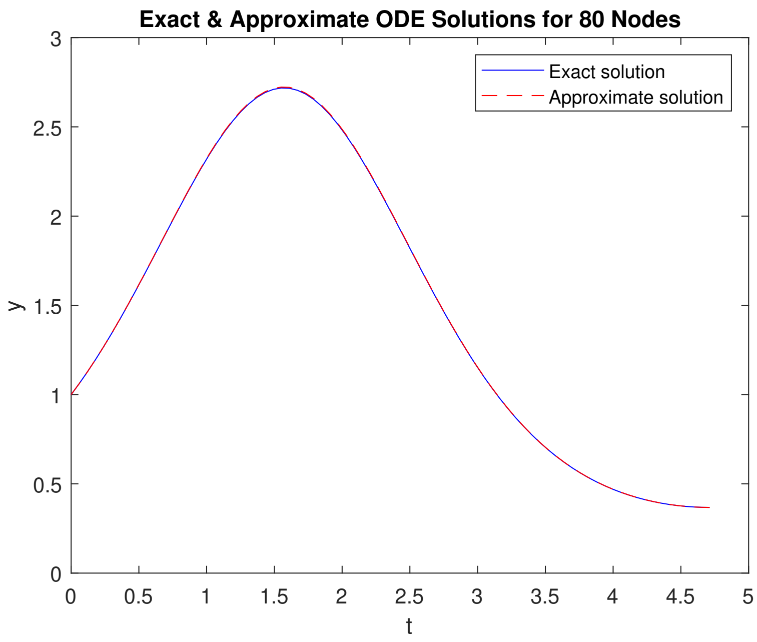

Figure 4: SOR-approximate and exact solution for 80 nodes.

Furthermore, the error between the approximate solution and the exact solution in the infinity norm as well as the error ratio $\frac{E_n}{E_{2n}}$ were calculated for different values of n. The optimal value of ω was also tested for the different number of nodes.

Table 1: Largest error and error ratio for different values of n.

n	$\|\|y_{exact} - y_{approx}\|\|_\infty$	Error Ratio	Optimal ω
10	0.4534	-	1.59
20	0.0879	5.1581	1.78
40	0.0204	4.3088	1.88
80	0.0049	4.1633	1.94

5 Discussion

The approximate solutions were plotted together with the exact ones and it was visually obvious that they followed the same trend, as predicted. The approximation also clearly increases in accuracy as more nodes are added. As indicated in the error ratio being approximately four in table 1, this method has second-order convergence, meaning when the number of nodes is doubled the error is divided by $2^2 = 4$. This makes sense since central finite difference has second-order accuracy. From table 1, it is also clear that the optimal value of ω increases as the number of nodes increases.

Decreasing the tolerance of the Successive Over-relaxation function to less than 10^{-11} and thus forcing it to iterate more times does not seem to significantly increase accuracy in the results. However, it does come with additional computational costs, which means that the program in Matlab will not be as fast. For example, decreasing the tolerance from 10^{-11} to 10^{-16} (machine precision) when using 80 nodes will increase the runtime from 0.04 seconds to 0.50 seconds while the error only decreases with 5.87×10^{-11}.

To increase the accuracy of the approximate method, an even higher order of accuracy stencil could be used when approximating the different derivatives. However, this improvement would also come with additional computational costs and the program in Matlab would not be as fast.

6 Program listing

```matlab
function [y_approx, error] = ProjectODE(num_nodes, tol)
% A Forward Euler method for solving the differential equation
% y'' = cos(x)y'-sin(x)y on the interval [0,3*pi/2] with the initial
% conditions y(0)=1 and y(3*pi/2)=exp(-1)
% Consider running the SOR_w_test.m after the first run to get the optimal
% w for your matrices
%
%   INPUT
%      num_nodes   - the number of nodes used for the approximation
%      tol         - user set tolerance for the stopping condition in the
%                    iteration
%
%   OUTPUT
%      y_approx    - n by 1 vector with the approximate function values to y
%      error       - the error between the exact and approximate function
%                    values calculated in the infinity norm

%   adds the Matrix Solver directory to the current frame
    addpath 'C:\Users\David Wiman\Documents\MATLAB\TANA21\Matrix Solver'

%   sets the interval, time step length, size of the system and allocates
%   space for A
    h = (3*pi)/(2*(num_nodes-1));
    x = [0:h:(3*pi)/2];
    n = length(x);
    A = zeros(n,n);

%   computes the exact function values
    y_exact = exp(sin(x))';

%   computes and inserts the correct elements into A on row 2 to n-1
    for i = 2:n-1
        alfa = (1/h^2)+(cos((i-1)*h)/(2*h));
        beta = (-2/(h^2))+sin((i-1)*h);
        gamma = (1/h^2)-(cos((i-1)*h)/(2*h));
        A(i,i-1:i+1) = [alfa beta gamma];
    end

%   inserts the correct elements into A for row 1 and n
    A(1,1) = 1;
    A(n,n) = 1;

%   creats the right hand side vector and inserts the correct elements
    b = zeros(n,1);
    b(1,1) = 1;
    b(n,1) = exp(-1);

%   solves the maxtrix system iteratively using the SOR method
    [y_approx,~] = SuccessiveOverRelaxation(A,b,1.95,ones(n,1),tol);

%   computes the error between the approximate and the exact solution in
%   the infinity norm
    error = norm(y_exact-y_approx, inf);

%   plots the exact and approximate solutions in the same graph to visually
%   confirm their resemblens
    plot(x,y_exact,'b');
    hold;
    plot(x,y_approx,'r--');
    title("Exact & Approximate ODE Solutions for 80 Nodes")
    xlabel("t")
    ylabel("y")
    legend("Exact solution","Approximate solution")

end
```

```
function [x, k] = SuccessiveOverRelaxation(A,b,w,x,tol)
%  Gauss-Seidel iterative method to approximate the solution of a
%  linear system Ax=b up to a user defined tolerance
%
%  INPUT:
%    A   - n by n square, non-singular, diagonally dominant matrix
%    b   - n by 1 right hand side vector
%    w   - 1 by 1 constant
%    x   - n by 1 vector containing that initial guess for the iteration
%    tol - user set tolerance for the stopping condition in the iteration
%
%  OUTPUT:
%    x - n by 1 vector containing the iterative solution
%    k - number of iterations

%  get the system size
   n = length(A);

%  defining the maximum number of iterations
   max_its = 22000;

%  looping through the iteration index k
   for k = 1:max_its
%  looping through the rows of the x vector
       for i = 1:n
           sum_1 = 0;
           sum_2 = 0;
%  creating the two sums needed in the formula
           for j = 1:i-1
                sum_1 = sum_1 + A(i,j)*x(j,k+1);
           end
           for j = i+1:n
               sum_2 = sum_2 + A(i,j)*x(j,k);
           end
%   updating the iterative solution
           x(i,k+1) = (1-w)*x(i,k)+(b(i)-sum_1-sum_2)*(w/A(i,i));
       end
%   checking if the tolerance i met and stopping if that's the case
       r = b - A*x(:,k);
       if norm(r,2) <= tol*norm(b,2)
           break
       end
   end

%   isolating the last column of x which contains the iterative solution
%   that met the stopping condition
    x = x(:,end);

end
```

```
function w = SOR_w_test(A,b,x,tol)
%  A test function to experimentally determine the optimal value of omega
%  for the Successive Over-relaxation method
%
%  INPUT:
%    A   - n by n square, non-singular, diagonally dominant matrix
%    b   - n by 1 right hand side vector
%    x   - n by 1 vector containing that initial guess for the iteration
%    tol - user set tolerance for the stopping condition in the iteration
%
%  OUTPUT
%    w   - the omega that results in the least number of iterations

%   allocates space for the vector containing the number of iterations for
%   each omega
    iteration_vector = zeros(1,101);

%   looping through the different omegas with 0.01 increments and computing
%   and storing the number of iterations needed to met the tolerance
    for w = 1:0.01:2
        [~,k] = SuccessiveOverRelaxation(A,b,w,x,tol);
        index = round((w-1)*100+1);
        iteration_vector(index) = k;
    end

%   finding the minimum number of iterations needed and finding the
%   corresponding omega
    [~,min_index] = min(iteration_vector);
    w = [1:0.01:2];
    w = w(min_index);

end
```