02685

Scientific Computing for Differential Equations

Assignment 3

Phillip Brinck Vetter (s144097)
Raja Shan Zaker Mahmood (s144102)
Steffen Sloth (s144101)

Group 1
April 26, 2018
Technical University of Denmark

Introduction

In this assignment we present answers to Assignment 3 of the course.. Information are gathered from course slides or the book [LeVeque, 2007] both cases will be evident in the text. Code will be included directly in the text where it is deemed to provide a better overview, while some functions are relegated to the appendix. We had trouble with the implementation of the interpolation function. While it is correctly implemented the elements are attempted to be naturally ordered which seems to lead to some differences. We troubleshooted with Group 2 (Rasmus and Asbjørn) of the course and their resulting function $interpolate_a.m$ was used. It is hence, identical and reproduced here with their permission. In section 3.3, we also present our own implementation.

Exercise 1: 2-point BVPs

Exercise 1.1 - Newton's method for solving nonlinear BVPs

Here we consider the nonlinear BVP given in eq. (2.91) in the book with Dirichlet boundary conditions.

$$\begin{aligned} \epsilon u''(x) + u(x)\left(u'(x) - 1\right) &= 0 \\ u(0) = \alpha \qquad u(1) &= \beta \end{aligned} \tag{1}$$

Ex. 1.1.1

Two derive a second order accurate scheme we chose to use central finite difference, which has the desired order of accuracy. It comes from combining the Taylor expansion of $u(x)$ around the points $\bar{x} \pm h$, where $\bar{x}$ is a point in the grid, and later h will be chosen as the step size.

$$u(\bar{x}+h) = u(\bar{x}) + hu'(\bar{x}) + \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5) \tag{2}$$

$$u(\bar{x}-h) = u(\bar{x}) - hu'(\bar{x}) + \frac{h^2}{2}u''(\bar{x}) - \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5) \tag{3}$$

where the superscripts denote the order of the derivatives.

To get an expression for the first order derivative, the term $u(x)$ gets isolated is eq. 3, and then inserted into eq. 2:

$$\begin{aligned} u(\bar{x}+h) &= \left(u(\bar{x}-h) + hu'(\bar{x}) - \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) - \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5)\right) \\ &\qquad + hu'(\bar{x}) + \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5) \\ &= u(\bar{x}-h) + hu'(\bar{x}) - \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) - \frac{h^4}{24}u^{(4)}(\bar{x}) \\ &\qquad + hu'(\bar{x}) + \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5) \\ &= u(\bar{x}-h) + 2hu'(\bar{x}) + \frac{h^3}{3}u'''(\bar{x}) + \mathcal{O}(h^5) \end{aligned} \tag{4}$$

and now the first order term $u'(x)$ get isolated in eq. 4:

$$\begin{aligned}
\Rightarrow 2hu'(\bar{x}) &= u(\bar{x}+h) - u(\bar{x}-h) - \frac{h^3}{3}u'''(\bar{x}) + \mathcal{O}(h^5) \\
\Rightarrow u'(\bar{x}) &= \frac{u(\bar{x}+h) - u(\bar{x}-h)}{2h} - \frac{h^2}{6}u'''(\bar{x}) + \mathcal{O}(h^4) \\
&= \frac{u(\bar{x}+h) - u(\bar{x}-h)}{2h} + \mathcal{O}(h^2) \\
&\approx \frac{u(\bar{x}+h) - u(\bar{x}-h)}{2h} \quad \text{(Approximation to first order derivative))}
\end{aligned} \tag{5}$$

A similar aproach can be used to approximate the second order derivative, by first isolating $u'(x)$ in eq. 3, inserting it into eq. 2, and then just isolating the second order term, as follows:

$$\begin{aligned}
u(\bar{x}+h) = u(\bar{x}) + \Big(u(\bar{x}) - u(\bar{x}-h) + \frac{h^2}{2}u''(\bar{x}) - \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5)\Big) \\
+ \frac{h^2}{2}u''(\bar{x}) + \frac{h^3}{6}u'''(\bar{x}) + \frac{h^4}{24}u^{(4)}(\bar{x}) + \mathcal{O}(h^5)
\end{aligned} \tag{6}$$

$$\begin{aligned}
&= 2u(\bar{x}) - u(\bar{x}-h) + h^2u''(\bar{x}) + \frac{h^4}{12}u^{(4)}(\bar{x})\mathcal{O}(h^5) \\
\Rightarrow h^2u''(\bar{x}) &= u(\bar{x}+h) - 2u(\bar{x}) + u(\bar{x}-h) + \frac{h^4}{12}u''''(\bar{x}) + \mathcal{O}(h^6) \\
\Rightarrow u''(\bar{x}) &= \frac{u(\bar{x}+h) - 2u(\bar{x}) + u(\bar{x}-h)}{h^2} + \mathcal{O}(h^2) \\
&\approx \frac{u(\bar{x}+h) - 2u(\bar{x}) + u(\bar{x}-h)}{h^2} \quad \text{(Approximation to second order derivative))}
\end{aligned} \tag{7}$$

Now using the two derived approximations for the first and second order derivatives an approximation to eq. 1 can be written as:

$$\begin{aligned}
\epsilon u''(x) + u(x)\left(u'(x) - 1\right) \approx \epsilon \left(\frac{u(\bar{x}+h) - 2u(\bar{x}) + u(\bar{x}-h)}{h^2}\right) \\
+ u(\bar{x})\left(\left(\frac{u(\bar{x}+h) - u(\bar{x}-h)}{2h}\right) - 1\right) = f(x)
\end{aligned} \tag{8}$$

which can be rewritten as a scheme of the from:

$$U_i = \epsilon\left(\frac{U_{i+1} - 2U_i + U_{i-1}}{h^2}\right) + U_i\left(\left(\frac{U_{i+1} - U_{i-1}}{2h}\right) - 1\right) \tag{9}$$

which is second order accurate due to the construction of the approximation to the first and second order derivative in eq. 5 and 7.

The local truncation error $\tau(x)$ is calculated, by inserting the true solution $u^*(x)$ into eq. 8 (and subtracting the right hand side f).

$$\tau(\bar{x}) = \epsilon\left(\frac{u^*(\bar{x}+h) - 2u^*(\bar{x}) + u^*(\bar{x}-h)}{h^2}\right) + u^*(\bar{x}-h)\left(\frac{u^*(\bar{x}+h) - u^*(\bar{x}-h)}{2h} - 1\right) - f(\bar{x}) \tag{10}$$

Equation 10 is not equal to zero due to the error from the approximations of the derivatives (from eq. 5 and 7). To rewrite equation 10 in term of derivatives and order of step size (h^n), the terms with $\bar{x} \pm h$ is Taylor expanded, as follows:

$$
\begin{aligned}
\tau(\bar{x}) &= \epsilon\left(\frac{u^*(\bar{x}+h)-2u^*(\bar{x})+u^*(\bar{x}-h)}{h^2}\right) \\
&\quad + u^*(\bar{x}-h)\left(\frac{u^*(\bar{x}+h)-u^*(\bar{x}-h)}{2h}-1\right)-f(\bar{x}) \\
&= \frac{\epsilon}{h^2}\,(\,\left(u^*(\bar{x})+hu^{*\prime}(\bar{x})+\frac{h^2}{2}u^{*\prime\prime}(\bar{x})+\frac{h^3}{6}u^{*\prime\prime\prime}(\bar{x})+\frac{h^4}{24}u^{*\prime\prime\prime\prime}(\bar{x})+\mathcal{O}(h^5)\right) \\
&\quad -2u^*(\bar{x})+\left(u^*(\bar{x})-hu^{*\prime}(\bar{x})+\frac{h^2}{2}u^{*\prime\prime}(\bar{x})-\frac{h^3}{6}u^{*\prime\prime\prime}(\bar{x})+\frac{h^4}{24}u^{*\prime\prime\prime\prime}(\bar{x})+\mathcal{O}(h^5)\right)\,) \\
&\quad + \frac{u^*(\bar{x})}{2h}\,(\,\left(u^*(\bar{x})+hu^{*\prime}(\bar{x})+\frac{h^2}{2}u^{*\prime\prime}(\bar{x})+\frac{h^3}{6}u^{*\prime\prime\prime}(\bar{x})+\frac{h^4}{24}u^{*\prime\prime\prime\prime}(\bar{x})+\mathcal{O}(h^5)\right) \\
&\quad -\left(u^*(\bar{x})-hu^{*\prime}(\bar{x})+\frac{h^2}{2}u^{*\prime\prime}(\bar{x})-\frac{h^3}{6}u^{*\prime\prime\prime}(\bar{x})+\frac{h^4}{24}u^{*\prime\prime\prime\prime}(\bar{x})+\mathcal{O}(h^5)\right)-2h\,)-f(\bar{x}) \\
&= \epsilon\left(u^{*\prime\prime}(\bar{x})+\frac{h^2}{12}u^{*\prime\prime\prime\prime}(\bar{x})+\mathcal{O}(h^4)\right) \\
&\quad + u^*(\bar{x})\left(u^{*\prime}(\bar{x})+\frac{h^2}{6}u^{*\prime\prime\prime}(\bar{x})+\mathcal{O}(h^4)-1\right)-f(\bar{x})
\end{aligned}
\tag{11}
$$

where the Taylor series in eq. 2 and 3 was used.

Now using the that $f(x)=\epsilon u''+u(u'-1)$ from the differential equation given in the assignment, the expression for the local truncation error ($\tau(x)$) in eq. 11 reduces to:

$$
\begin{aligned}
\tau(\bar{x}) &= \epsilon\left(\frac{h^2}{12}u^{*\prime\prime\prime\prime}(\bar{x})+\mathcal{O}(h^4)\right)+u^*(\bar{x})\left(\frac{h^2}{6}u^{*\prime\prime\prime}(\bar{x})+\mathcal{O}(h^4)\right) \\
&= h^2\left(\frac{\epsilon}{12}u^{*\prime\prime\prime\prime}(\bar{x})+\frac{u^*(\bar{x})}{6}u^{*\prime\prime\prime}(\bar{x})\right)+\mathcal{O}(h^4)=\mathcal{O}(h^2)
\end{aligned}
\tag{12}
$$

where it is clear that the highest order of h (the step size) is second order (h^2).

From eq. 12 it can be seen that the dominant term in τ is $\frac{\epsilon}{12}u^{*\prime\prime\prime\prime}(\bar{x})+\frac{u^*(\bar{x})}{6}u^{*\prime\prime\prime}(\bar{x})$, which depends both of the third and fourth order derivative and $u(x)$ itself.

Ex. 1.1.2

After defining the initial condition and the grid, eq. (2.106) in the book is used as an initial guess to the solution with an interior layers centered at $\bar{x}$ with the width ω_0, which is both given in the book on page 48.

Since the scheme in eq. 9 is implicit the system is solved using a Newton solver, where the expression to minimizes is eq. 8, which is set equal to zero by subtracting $f(x)$ on both sides. This gives the residual function called $G(U)$, eq. (2.106) in the book. The Matlab script to evaluate $G(U)$ can be seen in appendix A.1.
To estimate the change in U in each Newton iteration is the gradient of $G(U)$ estimated by use of the Jacobian. The Jacobian can be found by:

$$
J_{i,j}(U)=\frac{\partial}{\partial U_j}G_i(U) \tag{13}
$$

which for the first two rows in the Jacobian matrix gives:

$$\begin{aligned}
J_{1,1}(U) &= \frac{\partial}{\partial U_1}G_1(U) = \frac{-2\epsilon}{h^2} + \frac{U_2-\alpha}{2h} - 1\\
J_{1,2}(U) &= \frac{\partial}{\partial U_2}G_1(U) = \frac{\epsilon}{h^2} + \frac{U_1}{2h}\\
J_{2,1}(U) &= \frac{\partial}{\partial U_1}G_2(U) = \frac{\epsilon}{h^2} - \frac{U_2}{2h}\\
J_{2,2}(U) &= \frac{\partial}{\partial U_2}G_2(U) = \frac{-2\epsilon}{h^2} + \frac{U_3-U_1}{2h} - 1\\
J_{2,3}(U) &= \frac{\partial}{\partial U_3}G_2(U) = \frac{\epsilon}{h^2} + \frac{U_2}{2h}
\end{aligned} \tag{14}$$

Considering eq. 14 a common scheme for constructing the elements of the Jacobian would be:

$$J_{i,j}(U) = \begin{cases} \frac{\epsilon}{h^2} - \frac{U_i}{2h} & \text{if j=i-1}\\ \frac{-2\epsilon}{h^2} + \frac{U_{i+1}-U_{i-1}}{2h} - 1 & \text{if j=i}\\ \frac{\epsilon}{h^2} + \frac{U_i}{2h} & \text{if j=i+1}\\ 0 & \text{otherwise} \end{cases} \tag{15}$$

The Matlab code for evaluating the Jacobian can be seen in appendix A.2.

The final solver takes the shape seen in the following code:

```
clear all;
close all;
%% 2-point BVPs: Newton's method for solving nonlinear BVPs
alpha = -1; % Left boundary condition
beta = 1.5; % Right boundary condition
epsilon = 0.1; % Value for epsilon factor (epsilon << 1)
a = 0; % Starting point of interval
b = 1; % End point of interval
m = 100; % Number of grid points
h=(b-a)/(m+1); % Step size
X = a+h:h:b-h; % Grid
w0 = 1/2*(a-b+beta-alpha); % Width of the internal layer (omega_0)
xbar = 1/2*(a+b-beta-alpha); % Position of the center of the internal layer
maxit = 1000; % Maximum number of newton iterations
tol = 10^(-3); % Tolerance of Newton solver

uOuter(:,1) = [a,X,b]' + alpha - a; % Solution with only left boundary condition
uOuter(:,2) = [a,X,b]' + beta - b; % Solution with only right boundary condition
f = @(x) x - xbar + w0*tanh(w0*(x-xbar)/(2*epsilon)); % Initial guess (eq. 2.105)
U = f(X)'; % Set initial guess

% Newton solver
G = GEval(U,m,h,alpha,beta,epsilon); % Generate the residual for the system
J = JacEval(U,m,h,alpha,beta,epsilon); % Define the Jacobian
k=0; % Counter of Newton iterations
while ((norm(G,'inf') > tol) && (k <= maxit)) % Newton solver
    k=k+1; % Count Newton step
    dGdx = J; % Estimate the change in gradient of G
    dx = -dGdx\G; % Estimate the change in G
    U = U + dx; % Change the solution U
    J = JacEval(U,m,h,alpha,beta,epsilon); % Estimate the Jacobian
    G = GEval(U,m,h,alpha,beta,epsilon); % reevaluate the residual
    if k == maxit % Test if reached maximum number of iterations
        disp('Used max iterations !!!!!!')
    end
end
U=[alpha;U;beta]; % Add the boundary conditions to the solution
```

```
% Plot the results
figure;
subplot(1,2,ii)
hold on;
plot([a,X,b],uOuter(:,1),'k--','linewidth',2);
plot([a,X,b],uOuter(:,2),'k-.','linewidth',2);
plot([a X b],U,'r-*','linewidth',3)
fplot(f,[a b],'b-','linewidth',2)
title(strcat('Solution with \epsilon=',num2str(epsilon)))
legend('Solution at left boundary','Solution at right boundary',...
    strcat('Approximated solution (k=',num2str(k),')'),'Initial guess','location','northwest')
xlabel('x')
ylim([min(U) max(f(X))*1.2])
set(gca,'fontsize',20)
```

The Newton solver makes a change in U by solving $G(U)dx = dGdx = J$ for dx, while testing if the norm of $G(U)$ is less than the specified tolerance. The Newton solver estimates both the residual and Jacobian in each iteration.

The result of code applied to system (1) is shown in figure 1 for two value of ϵ.

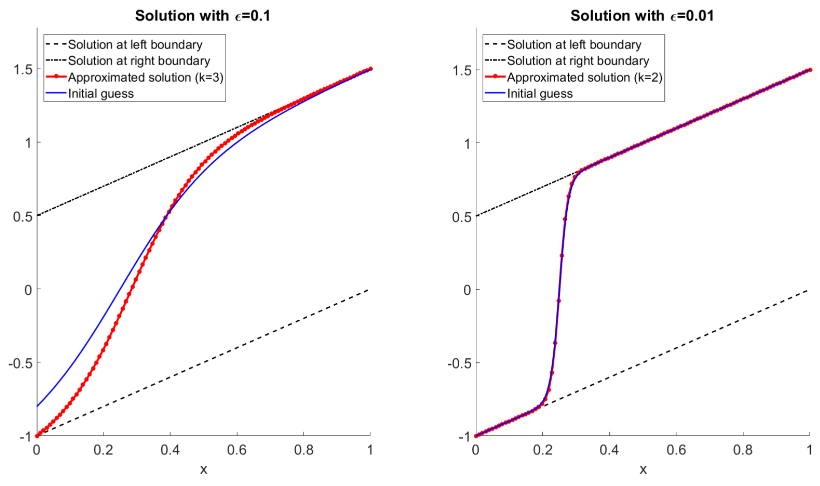


Figure 1: Solution for the system given in eq. 1 for two value of $\epsilon = [0.1; 0.01]$. The two black dotted lines indicates the solution for $\epsilon = 0$ with either the left or right boundary condition. The blue line in the initial guess given by eq. (2.105) in the book. The red line is the solution approximated by the scheme in eq. 8. The k-value in the legend shows the number of Newton iterations.

The results presented in figure 1 shows qualitatively good results, which approaches the analytic solution for the boundaries with $\epsilon = 0$ in the boundaries (goes toward the black dotted lines). The initial guess is good enough to converge to a solution after only a few ($k = [2, 3]$) Newton interactions.

Ex. 1.1.3

One approach could be to improve the scheme it self. This could be do by modifying the right hand side of the equation $AU = F$, to include e.g the leading term of the error so this term cancels with the dominating term of the error when applying the scheme. In eq. **??** we found the leading term of the error for this problem, which contains both the third and fourth order derivatives. This means that both the third and fourth order derivatives has to be know or a method to estimating them would be needed, which for this problem is not possible!

Another approach is to improve the effectiveness of the method. If the number of grid point stay the same, then using an nonuniform grid could improve the method. Having a higher density of grid point in regions with large changes in the function, would make the scheme better at predicting the change in this region(s). For this problem the region around the interior layer is where a higher density of grid point would be an advantage. With $0 < \epsilon \ll 1$ is the position ($\bar{x}$) and width of the transition region (ω_0) known from eq. (2.103) and eq. (2.104) in the book, which can be used to identify where on the grid more grid points is need, and how wide the region with an increase density of grid points should be.
One simple way to implement this could be to distribute the grid point using e.g a Gaussian or polynomial distribution, center at $\bar{x}$ and with width ω_0, to define the density of grid points over the region $a \leq x \leq b$.

Exercise 1.2 - Shooting Methods

Ex. 1.2.1

The problem can be rewritten into an IVP by introducing the two functions $x_1(t) = u(t)$ and $x_2(t) = x_1'(t) = u'(t)$. Through isolation of the second derivative $u''(t)$ in (1) one obtains

$$u''(t) = \frac{u(t)\left[1 - u'(t)\right]}{\epsilon} \tag{16}$$

from which the system can be derived as

$$\mathbf{x}'(t) = \begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \mathbf{f}(\mathbf{x}(t)) = \begin{bmatrix} x_2(t) \\ \dfrac{x_1(t)\left[1 - x_2(t)\right]}{\epsilon} \end{bmatrix} \qquad \mathbf{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} \alpha \\ \sigma \end{bmatrix} \tag{17}$$

Ex. 1.2.2

The objective is then to solve the system for σ given in (17) such that the end value of the solution is given by $\mathbf{x}(t_{end}) = \mathbf{x}(1) = \beta$. The problem can be formulated by

$$G(\sigma) = x_{1,\sigma}(1) - \beta = u_\sigma(1) - \beta = 0 \tag{18}$$

The system is solved using a Newton's Method approach. This method was chosen as it was anticipated due to the inclusion of the sensitivities that the method would provide faster convergence at a higher tolerance than the other simpler methods. Newton's Method is employed through a Taylor expansion which yields an expression for the next step of an iteration sequence. The process can be derived by considering the expansion of $G(\sigma)$ as follows:

$$G(\sigma_{k+1}) = G(\sigma_k) + \nabla G(\sigma_k)^T(\sigma_{k+1} - \sigma_k) = 0 \quad \Rightarrow \quad \underline{\sigma_{k+1} = \sigma_k - \left[\nabla G(\sigma)\right]^{-T} G(\sigma_k)} \tag{19}$$

In this case where $G(\sigma)$ is a scalar function the Jacobian simply reduces to the derivative $G'(\sigma)$. Notice then from (18) that

$$G'(\sigma) = \frac{d}{d\sigma}\left[u_\sigma(1) - \beta\right] = \left.\frac{du_\sigma}{d\sigma}\right|_\sigma = \left.\frac{dx_1}{d\sigma}\right|_\sigma \tag{20}$$

The derivative necessary for the implementation of the Newton's Method is the derivative of the solution with respect to the initial value. This derivative can be computed by solving the sensitivity differential equation

$$S'(t) = \frac{d\mathbf{f}}{d\mathbf{x}}S(t) \qquad S(t_0) = I \tag{21}$$

where the derivative of the right hand side **f** with respect to **x** is given by

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \frac{df_1}{dx_1} & \frac{df_1}{dx_2} \\ \frac{df_2}{dx_1} & \frac{df_2}{dx_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{1-x_2}{\epsilon} & \frac{x_1}{\epsilon} \end{bmatrix} \tag{22}$$

Fortunately this can easily be achieved simultaneously with solving (17). This is in practice carried out by transforming the matrix $S'(t)$ into a column vector such that the vectors of the IVP simply contain additional inputs. The differential equation can then be solved using a previously developed IVP-solver. The one used here was the DormandPrince5(4).

The script below is the solution to the problem where the function pointer @IVP is provided in Appendix A.3.

```
%% Exercise 1.2 Shooting Methods
clear variables; close all; clc;

alpha = -1;
beta = 1.5;
epsilon = 0.1;

%Initial Guess
sigma = 1.7656;     %Solution VERY sensitive to starting guess for eps < 0.5
I = eye(2);
x0 = [alpha ; sigma ; I(:)];

%Conditions
tspan = [0 1];
h = 10^(-3);

%Converged Tolerances
tol = 10^(-2);
maxit = 20;
k = 0;
abstol = 10^(-3);
reltol = 10^(-3);

Converged = false;
while ~Converged
    if k >= maxit
        disp('Maximum iterations reached. Aborted')
        break
    end
    k = k + 1;
    [T,X] = DormandPrinceSolver(@IVP, tspan, x0, h, abstol, reltol, epsilon);
    %[T,X] = ode45(@IVP,tspan,x0,[],epsilon);

    %Update Sigma (The sixth element in column X is the desired derivative)
    G_sigma = X(end,1) - beta;
    dxdG_sigma = X(end,5);
    ds = G_sigma/dxdG_sigma;

    sigma = sigma - ds;
    x0(2) = sigma;

    %Check Convergence
    Converged =  norm(G_sigma) <= tol ;
    if Converged
        sprintf('The minimum was succesfully found after %d steps. \n The value of X(b) = %f and sigma = %
        stat.sigma = sigma;
        stat.sens = dxdG_sigma;
    else
        sprintf('The current step no. %d has value \n X(b) = %f , sigma = %f',k,X(end,1),sigma)
    end
end

%When choosing epsilon certainly low there is problems with the step.
%We sometimes get the following messages while sigma and X(b) blows up

%"Warning: Failure at t=5.311012e-01.
%Unable to meet integration tolerances without reducing the step size
%below the smallest value allowed (1.776357e-15) at time t."
```

The solution with a tolerance $T = 10^{-4}$ is then found to be

$$\sigma_{opt} \approx 1.7659 \tag{23}$$

The solution was not easily found for the given value of $\epsilon = 0.1$ however. The Newton's Method in its implemented non-corrected form lacks tools to provide robustness. The initial guess had to manually be narrowed down such that in the end the solution was in fact the initial guess which is evident at a closer glance at the provided script. Additional discussion relating this is provided in **Ex. 1.2.4**. A plot of the solution $u_\sigma(t)$ with the found σ corresponding solution in the limit $\epsilon \to 0$ in provided below in Figure (2)

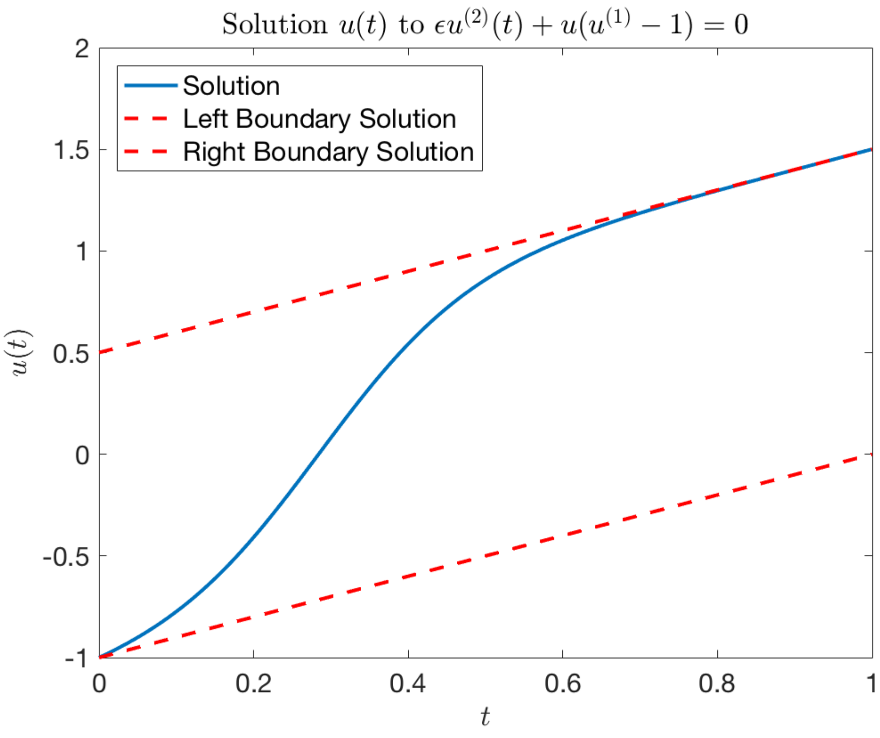


Figure 2: Solutiong using the implemented Shooting method. The two red dotted lines are the solution to the differenetial equation for $\epsilon = 0$ with either the left or right boundary condition.

Ex. 1.2.3

The sensitivity at the final time for the provided solution and value of σ is found to be

$$\left.\frac{du_\sigma}{d\sigma}\right|_{t=1} \approx -36.0637 \tag{24}$$

It is evident then that the solution is highly affected by the value of σ in the final point judging by this numerically large value which seems to hint at the rather unstable environment surrounding converging towards the proper minimum of σ. Since the sensitivity exhibits a huge gradient at the solution it is

Ex. 1.2.4

As ϵ becomes very small the Newton's Method is unable to converge as it is often seen with such simple methods that are not guided by either a Trust-Region, Line-Search or similar approach. Typically one would also required a suitable initial guess based on some theoretical insight as briefly discussed in relation to the derived (2.105) [LeVeque, 2007]. Implementation of either one of such two methods could potentially fix the issue. Testing suggested that problems occurred already for $\epsilon < 0.5$ in which case convergence was not achieved. The issue that arises when ϵ becomes sufficiently small is assumed to be equivalent to that discussed through the *Péclet number* P.44 [LeVeque, 2007]. The transition between the two boundary solutions as depicted in Figure (2) tends towards a delta function (at least approximately for a very small ϵ) which increases the degree of non-linearity here. It is very likely to be part of the explanation as to why the Newton solver exhibits convergence issues.

Exercise 2: 9-point Laplacian

The first equation we want to solve with a numerical method implementing the 9-point Laplacian, is:

$$u_{0,exact}(x,y) = \sin(4\pi(x+y)) + \cos(4\pi xy) \tag{25}$$

which is a function of two variables x and y.

The desired equation to is the Possion equation with Dirichlet boundary conditions, which here takes the form:

$$\begin{aligned}\triangledown^2 u_{0,exact}(x,y) = 16\pi \Big(&\cos\left(4\pi\left(x^2+y^2\right)\right) \\ &- \pi\left(x^2+y^2\right)\left(4\sin\left(4\pi\left(x^2+y^2\right)\right) + \cos\left(4\pi xy\right)\right) \Big) = f(x,y)\end{aligned} \tag{26}$$

The linear system to solve is as usual:

$$A_9 U = F \tag{27}$$

where U is the solution, F is the right hand side of equation 26 together with the boundary conditions, and A_9 is the A-matrix for the 9-point Laplacian coming from the scheme given in eq. (3.17) in the book.

Since the A-matrix for the 9-ponit Laplacian is supplied (in the Matlab code *poisson9*), can we go straight into constructing the right hand side F. This is done in the code *form_rhs9*, which takes the inputs; number of grid points (m), the right hand side of equation 26 (f) evaluated in the grid points, and the boundary condition (bc).

One important thing to note is how we have constructed the boundary conditions, which in our code is saved in a $(m+2\times m+2)$-matrix. All the inner elements of the matrix is zero, and the "edge" of the matrix contains the values of $u_{0,exact}(x,y)$ (eq. 25) evaluated in the respective x,y-coordinates.
The code used to form the right hand side of equation 27 (*form_rhs9*) is shown in the following.

```
function F = form_rhs9(m,f,bc) % Forming right hand side of AU=F
    h = 1/(m+1); % grid point size
    bc = bc/(6*h^2); % Normalize to prefactor of scheme
    F = f; % initialize F
    % Corner element at (x,y)=(0,0)
    F(1) = F(1) - bc(1,1)-4*bc(1,2)-bc(1,3)-4*bc(2,1)-bc(3,1);
    % Corner element at (x,y)=(1,0)
    F(m) = F(m) - bc(1,m+2)-4*bc(1,m+1)-bc(1,m)-4*bc(2,m+2)-bc(3,m+2);
    % Corner element at (x,y)=(0,1)
    F(1+m*(m-1)) = F(1+m*(m-1)) -...
                   bc(m+2,1)-4*bc(m+2,2)-bc(m+2,3)-4*bc(m+1,1)-bc(m,1);
    % Corner element at (x,y)=(1,1)
    F(m^2) = F(m^2) -...
             bc(m+2,m+2)-4*bc(m+2,m+1)-bc(m+2,m)-4*bc(m+1,m+2)-bc(m,m+2);
    % Edge elements at x=[0;1] and y=0
    F(2:m-1,1) = F(2:m-1,1) -...
                 bc(1,2:m-1)'-4*bc(1,3:m)'-bc(1,4:m+1)';
    % Edge elements at x=0 and y=[0;1]
    F([m+1:m:1+(m-2)*m],1) = F([m+1:m:1+(m-2)*m],1) -...
                             bc(2:m-1,1)-4*bc(3:m,1)-bc(4:m+1,1);
    % Edge elements at x=1 and y=[0;1]
    F([2*m:m:m*(m-1)],1) = F([2*m:m:m*(m-1)],1) -...
                           bc(2:m-1,m+2)-4*bc(3:m,m+2)-bc(4:m+1,m+2);
    % Edge elements at x=[0;1] and y=1
    F(2+m*(m-1):(m-1)*(1+m)) = F(2+m*(m-1):(m-1)*(1+m)) -...
                               bc(m+2,2:m-1)'-4*bc(m+2,3:m)'-bc(m+2,4:m+1)';
end
```

The right hand side of equation 27 has to be a $m^2 \times 1$-vector initially set to equal f, here f has be straight out following the natural row-wise order. The right hand side (F) is then adjusts, to account for the boundary condition simply by subtracting them. This operation of course only performed on points next to the boundary, since the 9-point Laplacian uses all the eight closest neighbor points, as shown in figure 3.1,b in the book. It is important to remember that the neighbors weighted differently as also indicated in figure 3.1 and the scheme given by eq. (3.17).

The above code calculates firstly the step size (h) using the number of grid points (m), to be used for normalizing the boundary values to the scheme by multiplying with the prefactor of eq. (3.17) in the book.
Afterwards the right hand side (F) is initialize to equal the right hand function (eq. 26) evaluated on the grid. Here after is the four corner of the grid accounted for, which each has five boundary component. Lastly the four edges is modified, where each elements has three boundary components either to the right, left, above, or below dependent on the specific edge.
This finishes the correction need to the right hand side for the system to account for the boundaries.

Section 3.5 in the book discusses how to achieve high order of convergence by during an additional modification to the right hand side. The scheme can obtain high order of convergence by subtraction the dominating term in the error, since the error generated by the scheme should cancel with this additional term. In the case for the Possion equation is the domaniting term in the error $\bigtriangledown^4 u(x, y)$, which luckly is equivalent to the Laplace operator on the right hand function, $\bigtriangledown^2 f(x, y)$, here estimated using the 5-point Laplacian of the known right hand function $\bigtriangledown_5^2 f(x, y)$. This changes the initial value for the right hand side (before accounting for boundaries) to:

$$f_{ij} = f(x_i, y_j) + \frac{h^2}{12} \bigtriangledown^2 f(x_i, y_j) \approx f(x_i, y_j) + \frac{h^2}{12} \bigtriangledown_5^2 f(x_i, y_j) \tag{28}$$

where the second term is the dominating term in the error of the 9-point Laplacian scheme for the Possion equation.
This make the scheme fourth order accurate, since the following term in the error depend on h^4.

To estimate the 5-point Laplacian of the right hand function given in equation 26, the following scheme is used:

$$\bigtriangledown_5^2 f(x_i, y_j) = \frac{1}{h^2} \left(f(x_{i-1}, y_j) + f(x_{i+1}, y_j) + f(x_i, y_{j-1}) + f(x_i, y_{j+1}) - 4f(x_i, y_j) \right) \tag{29}$$

The following code is used to implement equation 29, which takes the inputs; number of grid points (m), the right hand function $f(x, y)$, and then the grid with boundary points included.

```
function LapRhs = LaplacianOfRhs(m,h,fun,bcX,bcY) % 5-point Laplacian of right hand function
    Lap5fun = zeros(m,m); % Initialize (m x m)-matrix
    for ii = 1:m % loop over rows (from bottom to top)
    for jj = 1:m % loop over colubms (from right to left)
        % Evaluate fun (right hand function) using the 5-point Lapacian scheme
        Lap5fun(ii,jj) = (fun(bcX(ii,jj+1),bcY(ii,jj+1))+fun(bcX(ii+1,jj),bcY(ii+1,jj))+...
                          fun(bcX(ii+1,jj+2),bcY(ii+1,jj+2))+fun(bcX(ii+2,jj+1),bcY(ii+2,jj+1))-...
                          4*fun(bcX(ii+1,jj+1),bcY(ii+1,jj+1)))/h^2;
    end
    end
    LapRhs = zeros(m^2,1); % Initialize (m^2 x 1)-vector as result
    for ii = 1:m % Reshape to a (m^1 x 1)-vector using the natural rowwise order
        LapRhs(1+(ii-1)*m:ii*m,1) = Lap5fun(ii,:);
    end
end
```

After generating the 5-point Laplacian of the right hand function as a $((m+2)\times(m+2))$-matrix, it gets reshaped into a $(m^2 \times 1)$-vector following the natural rowwise order.

The right hand function is then modified by adding the additional term, so:

$$f_{new} = f_{old} + \frac{h^2}{12}\,\nabla_5^2\, f(x_i, y_j) = f_{old} + \frac{h^2}{12} f_{5pointLap.} \tag{30}$$

where f_{new} is the new modified right hand function, f_{old} is the previous right hand function evaluated on the grid, and $f_{5pointLap.}$ is the 5-point Laplacian on the right hand function.

By implementing the modification in this way, no change is need for crating the right hand side (F), described above, only now the new right hand function is instead inserted into the Matlab function *form_rhs9*.

To solve the Possion equation (eq. 26) for the system with the exact solution given in equation 25, the following code is implemented.

```
%% 9-point Laplacian, fourth order accurate
a = 0; b = 1; c = 0; d = 1; % Define region
m = 100; % number of grid points
n = m+2; % number of grid points with boundaries
h = 1/(m+1); % grid point size
[X,Y] = meshgrid(a+h:h:b-h,c+h:h:d-h); % Define grid without boundaries
[bcX,bcY] = meshgrid(a:h:b,c:h:d); % Define grid with boundaries
u0 = @(x,y) sin(4*pi*(x.^2 + y.^2)) + cos(4*pi*x.*y); % Define exact solution
uTrue = u0(bcX,bcY); % Define true solution
border = [ones(1,n);[ones(n-2,1),zeros(n-2,n-2),ones(n-2,1)];ones(1,n)]; % Define border
% Define right hand function
fun = @(x,y) 16*pi.*(cos(4*pi*(x.^2+y.^2))-...
                pi.*(x.^2+y.^2).*(4.*sin(4*pi*(x.^2+y.^2))+cos(4*pi*x.*y)));
for ii = 1:m % Loop over each row to generated the old right hand funtion
    fOld(1+(ii-1)*m:ii*m,1) = fun(X(ii,:),Y(ii,1));
end
% Estimates correction term via 5-point Laplacian
LapRhs = LaplacianOfRhs(m,h,fun,bcX,bcY);
f = fOld + h^2/12*LapRhs; % Corrects for the domination term in error

bc = u0(bcX,bcY).*border; % Define baoundary at border
bcEdge = bc; % Define the edge as the boundary
bcEdge(bc == 0) = NaN; % Remove every other point than the bounda

A9 = poisson9(m); % Call code to generate A-matrix
F = form_rhs9(m,f,bc); % Call code to generate right hand side

U = A9\F; % Calculate the solution U (as a vector)
for ii = 1:m % Reshape U as a m x m matrix
    UU(ii,:) = U(1+(ii-1)*m:ii*m,1);
end
```

Firstly the grid, number of grid points, the step size, the right hand function, and the boundary conditions is defined.
Here after the right hand side (F) is constructed using the code *form_rhs9*, with the modification discussed just above to achieve high order of convergence, and the A-matrix is constructed using the given code *poisson9*.
To solve the system the backslash-command in Matlab is used $(U = A9\backslash F)$, which then afterwards gets reshaped from a $(m^2 \times 1)$-vector into a $(m \times m)$-matrix shaped after the natural rowwise order.

Plotting the result of the above method is shown in the following figure 3.

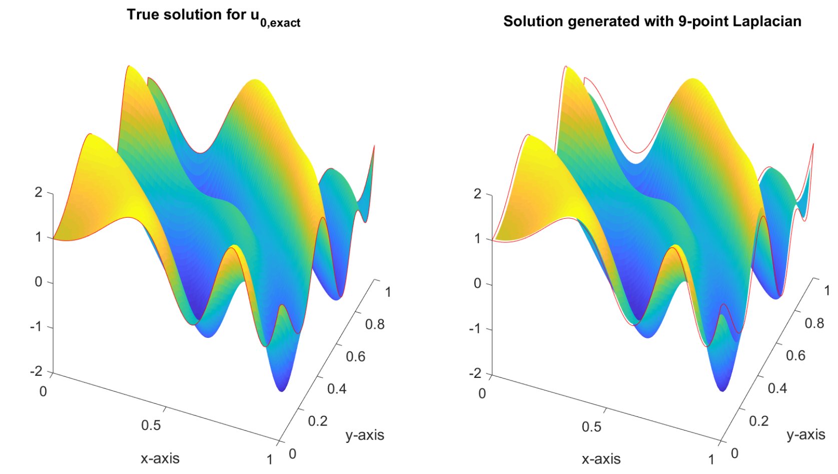


Figure 3: Solution to the 2D differential equation in eq. 26. To the left is the exact solution to the differential equation $u_{0,exact}$ (eq. 25). To the right is the solution generated using a 9-point Laplacian scheme, with the correction to the right hand side (F) given in eq. 29 and 30. The red line along the edge of both plots is the boundary condition. Note the surface plot of the approximated solution to the right does not contain the boundary condition, which is why it does not align with the red line (the boundary).

The solution generated by the 9-point Laplacian scheme can be seen to qualitatively match the true solution in figure 3. To best illustrate the approximated solution to the boundary condition, the solution is shown as only a $(m \times m)$-matrix not including the boundaries. this clearly shows that solution follows the boundaries along all four edges of the region.

To test if the scheme truly follows a fourth order convergence in step size, as would be expected with the correction in eq. 30, the code is run for several step sizes, and the error is estimated using the infinity norm.

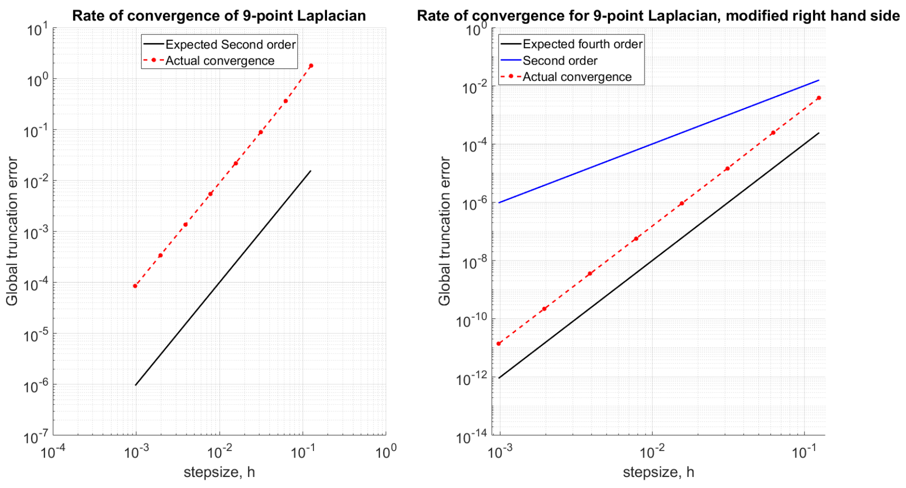


Figure 4: Logarithmic plots of rate of convergence using the 9-point Laplacian scheme on the Possion equation with the solution given in eq. 25. The left plot shows the rate of convergence for the "standard" 9-point Laplacian, which follows a second order line as expected. The right plot shows the order of convergence for the 9-point Laplacian scheme with a modified right hand side, which follows a fourth order line as expected.

The rates of convergence shown in figure 4 clearly indicates that we get the expected orders of convergence. The plot to the left in figure 4, shows that the "standard" 9-point Laplacian scheme, without the modified right hand side, follows a second order convergence as expected from equation following eq. (3.17) in the book. The plot to the right in figure 4 shows that the scheme follows a fourth order convergence rate, with the modified right hand side given by eq. 30.

Now the problem list two other solutions for the Possion equation to solve using this 9-point Laplacian solver, which are:

$$\begin{aligned} u_{1,exact}(x,y) &= x^2 + y^2 \\ u_{2,exact}(x,y) &= \sin\left(2\pi\left|x-y\right|^{2.5}\right) \end{aligned} \tag{31}$$

The right hand function to the two equations is:

$$\begin{aligned} f_1(x,y) &= \bigtriangledown^2 u_{1,exact}(x,y) = \bigtriangledown^2\left(x^2+y^2\right) = 4 \\ f_2(x,y) &= \bigtriangledown^2 u_{2,exact}(x,y) \\ &= 47.123\sqrt{|x-y|}sign\left(|x-y|\right)^2\cos\left(2\pi\left|x-y\right|^{2.5}\right) \\ &- 493.480\left|x-y\right|^3 sign\left(|x-y|\right)^2\sin\left(2\pi\left|x-y\right|^{2.5}\right) \end{aligned} \tag{32}$$

Note that the equation for $f_2(x,y)$ is not continues across the diagonal where $x = y$, which is why the Matlab sign-command comes into the equation. The sign-command i defined as ± 1 in $\pm arg \neq 0$, where arg is the argument, and zero when $arg = 0$! This is of course an approximation to the actual derivative in line of divergence $x = y$.
Using the Matlab sign-commands means we approximated the derivative to be zero along the diagonal $x = y$, which seems as a reasonable assumption since we know that that the solution $u_{2,exact}$ is zero along this diagonal, see figure 5a.
Other methods do also exist to estimate the derivative in the diagonal, e.g. using a stencil taking points from both sides of diagonal to estimate mean values along the diagonal, our approach is used because it is much simpler to implement.

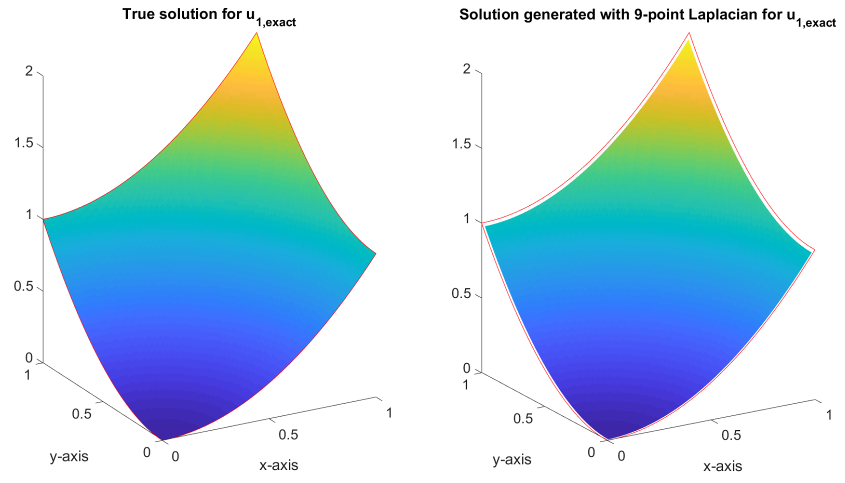
Applying the 9-point Laplacian solver to solve the Possion equation, with the two exact solutions given in eq. 31 with the right hand functions defined in eq. 32, gives the results shown in figure 5. Both the approximated solution in the right plots seems to look like the true solution in the left plots. Again is the boundaries not include in the approximated solution.

Again to test the rate of convergence, with this 9-point Laplacian solver which should be fourth accurate, has the solver been run with various step sizes and the error estimated using the infinity norm.

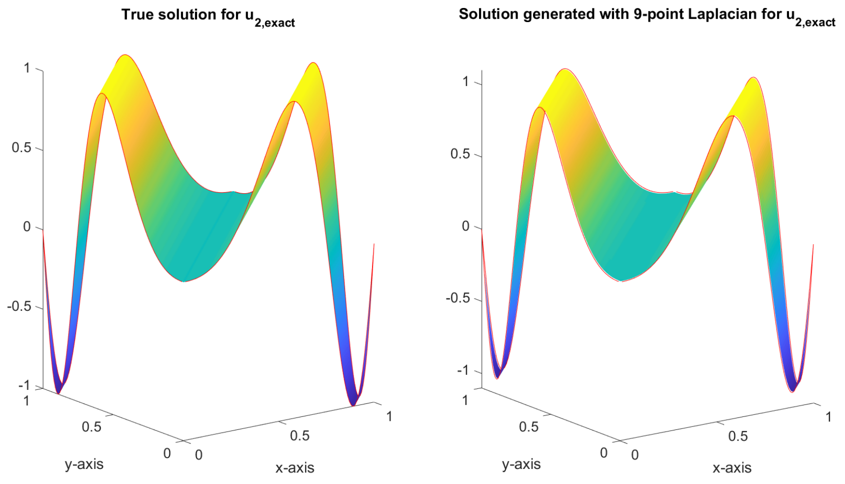
Figure 6 reveals that the rate of convergence does not follow the assumed fourth order behavior.

Figure 6a shows that the rate of convergence for $u_{1,exact}$ follows a zeros order. This is because the solution is a second order polynomial in x and y, so the third order derivative and upwards all equal zero. Since the dominating term in the error depend on derivatives from fourth order and upward, which all is zero, does the scheme actually becomes exact for this problem. Considering the parameters in the code one will also notice that the correction term for the right hand side becomes zero vector.
The reason why the rate of convergence in figure 6a is not completely flat is due to round-off errors, which we will not go into further details with here.

Figure 6b indicates that the solutions to $u_{2,exact}$ have a rate of convergence following an order of 3/2. To explain this one must consider the derivation of the scheme and the error of the scheme, shown in section 3.5 in the book. Both these derivations strongly depend on the fact that the desired solution/function in question is smooth, which means that any order of derivative is



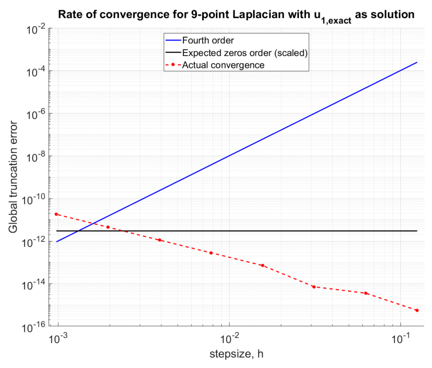
(a) System with the exact solution to the Possion equation being $u_{1,exact}$ in eq. 31.



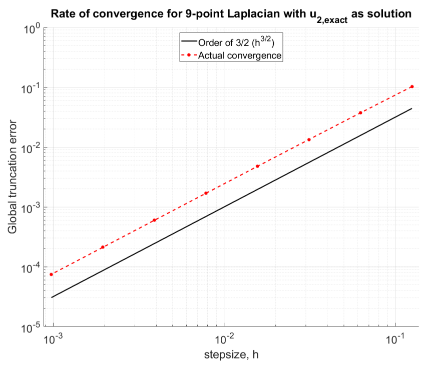
(b) System with the exact solution to the Possion equation being $u_{2,exact}$ in eq. 31.

Figure 5: Solutions to the 2D differential equation in eq. 26 to two different systems. The left plots is the exact solution to the differential equation $u_{1,exact}$ and $u_{2,exact}$ (eq. 31), respectively. The right plots is the solution generated using a 9-point Laplacian scheme, with the correction to the right hand side. The red line along the edge of both plots is the boundary condition. Note the surface plots of the approximated solutions to the right, does not contain the boundary conditions, which is why they do not align with the red line (the boundary).

defined everywhere in the domain, if this is not the fact no exact Taylor series can be found to the functions! Here the smoothness breaks down already at the first order derivative, where the derivative becomes undefined for the diagonal $x = y$. So the discretization using Taylor series leading to equation (3.17) in the book for the scheme and the following equations for the error, all fail for this problem since no such Taylor series exists!
To shows that the actual rate of convergence becomes 3/2, is however more difficult to explain. It is clear that it must be lower than second order, but the actual rate of convergence depends on how the derivative is approximated in the undefined diagonal.



(a) Rate of convergence for 9-point Laplacian used on the system with the exact solution to the Possion equation being $u_{1,exact}$ (eq. 31).



(b) Rate of convergence for 9-point Laplacian used on the system with the exact solution to the Possion equation being $u_{2,exact}$ (eq. 31).

Figure 6: Plots of the rate of convergence when using the 9-point Laplacian scheme with a modified right hand side. The black lines is the estimated order of convergences, and the red lines is the actual rate of convergences.

Exercise 3: Iterative solvers in 2D

Exercise 3.1 - Matrix-free 5-point Laplacian

With vector-input U number of interior points along one dimension m, we can construct the product $-A^hU$ without constructing the matrix. A^h is the 5−point discretization of the 2D Laplacian. The following MATLAB function demonstrates the implementation.

```
function AU = Amult(U,m)

%Initial Values
h = 1/(m+1);
L = length(U);

%Construct Grid
I = zeros(m+2);

%Construct Inner Matrix
%UM = reshape(U,m,m)';
UM = rot90(reshape(U,m,m));

%Embed Inner Matrix
I(2:m+1,2:m+1) = UM;

%Construct Laplacian Approx (Negative Sign)
AU_Temp = (1/(h^2))*(4*UM - I(2:m+1,3:m+2) - I(2:m+1,1:m) - I(3:m+2,2:m+1) - I(1:m,2:m+1));


%Transform back to a vector
AU = reshape(rot90(AU_Temp,3),L,1);
end
```

In order the verify the implementation the problem was attempted solved by using the PCG command in MATLAB. The command employs the Conjugate-Gradient Method. The method is a *direct method* in the sense that is converges within a finite number of steps which can be shown to be bounded by the number of distinct eigenvalues. Through an initial guess u_0 a solution is obtained after some iterations. The script developed for this procedure can be seen below. The algorithm converged in five iterations and the solution given by

$$\begin{bmatrix}5.0676 & -1.2056 & 12.3169 & -0.2504 & -2.7471 & -1.2056 & 2.1802 & -0.2504 & 5.0676\end{bmatrix}^T$$

was confirmed to match a solution obtained for a problem similar to that presented in **Exercise 2** (26) but for the 5-point Laplacian instead.

```
%% Solve -A*U = -F with PCG (Preconditioned Conjugate Gradient Method)

%Initialize. Function Handles and Interior Points
m = 3;
fun = @(x,y) 16*pi.*(cos(4*pi*(x.^2+y.^2))-pi.*(x.^2+y.^2).*(4.*sin(4*pi*(x.^2+y.^2))+cos(4*pi*x.*y)));
u0 = @(x,y) sin(4*pi*(x.^2 + y.^2)) + cos(4*pi*x.*y);

%Construct the Right-Hand Side
F = -form_rhs(fun,u0,m);

%Solve the Problem using PCG
U = zeros(1,9);                    %Initial Guess
J = @(U) Amult(U,m);               %Function Handle
[U_Sol,~,~,Iter,~] = pcg(J,F);    %Call PCG-Matlab Function
```

Exercise 3.2 - Under/over-relaxed Jacobi smoothing

From section 3.4, we find that the eigenvalues for the 2D Poisson problem, using the 5-point stencil, takes the following value as a function of $p, q = 1...m$, where m^2 is the number of grid points.

$$\lambda_{pq} = \frac{2}{h^2}\left((\cos(p\pi h) - 1) + (\cos(q\pi h) - 1)\right) \tag{33}$$

For the Jacobi method, the iteration matrix can be found by simple methods as.

$$\begin{aligned} G &= I - D^{-1}A & (34)\\ &= I + \frac{h^2}{4}A & (35)\end{aligned}$$

The eigenvalues of the iteration matrix, we call γ_{pq}.

$$\begin{aligned} \gamma_{pq} &= 1 + \frac{h^2}{4}\lambda_{pq} & (36)\\ &= 1 + \frac{1}{2}\left((\cos(p\pi h) - 1) + (\cos(q\pi h) - 1)\right)) & (37)\\ &= \frac{1}{2}\left(\cos(p\pi h) + \cos(q\pi h)\right) & (38)\end{aligned}$$

The iteration matrix for the underrelaxed Jacobi method is found in Leveque, chapter 4.6.

$$G_\omega = (1-\omega)I + \omega G \tag{39}$$

Giving eigenvalues that are simply stated, as a function of the relaxation parameter ω.

$$\gamma_{\omega pq} = (1-\omega) + \omega\gamma_{pq} \tag{40}$$

```
% problem size and spacing
m = 100;
h = 1/(m+1);
lambda = zeros(m,m);


% Nested for-loops evaluating the eigenvalues for the 2D-Poisson problem
for p = 1:1:m
    for q = 1:1:m
        lambda(p,q)  = 1+(h^2/4)*(2/h^2).*((cos(p*pi*h)-1)+(cos(q*pi*h)-1));
    end
end

% Only considering the higher frequency eigenvalues
lambda2 = lambda(m/2+1:end,m/2+1:end);

% We take omega in [0,2] and preallocate
omega = 0:0.01:2;
index = zeros(1,length(omega));

% Maximum absolute eigenvalue for each value of omega.
i = 1;
for k = omega
    temp=max(abs((1-k).*eye(length(lambda2))+k.*lambda2));
    index(1,i)=max(temp);
    i = i+1;
end

figure(1)
plot(omega,index)
xlabel('omega')
ylabel('max(gamma_{pq})')
```

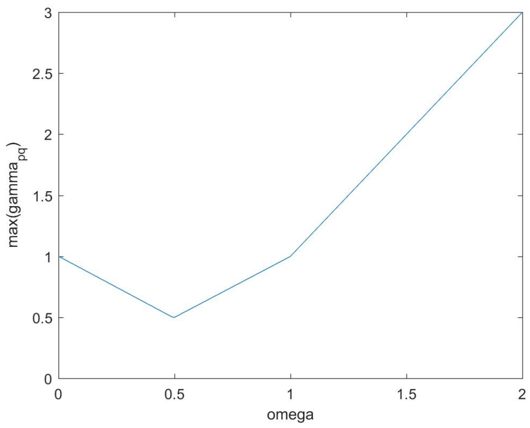


Figure 7: The maximum absolute value of the eigenvalues in the 2D Poisson prolem, as a function of ω. By visual inspection, the minimum is found at 0.25.

Graphically it is seen that the optimal value of ω is found when it has a value of about 0.5. We incorporate this information into a smoothing step. This step is an application of the underrelaxed Jacobi-method, taking one iteration of the method per function call. The update of the iteration is given from 4.88 in Leveque. We utilize that we have a matrix-free implementation of the crucial matrix product, implemented in section 3.1.

```
function Unew = smooth2(U,omega,m,F)

h = 1/(m+1);
Unew = U + omega * ( (h^2/2)*Amult(U,m) - (h^2/4)*F);

end
```

Exercise 3.3 - Multigrid solver

The Multigrid method in 2D is a direct extension of the 1D case. The rationale is the same: We are able to reduce the error of lower frequencies by changing the coarseness of the grid on which the function is approximated. The main difficulty arises when having to change between different sizes of grids in 2D. In order to achieve this mapping back and forth two functions are needed. One maps from a current grid to a coarser one while the other performs the inverse operation and interpolates from the coarsen version back onto the previously finer separated grid. The two desired functions have been implemented as shown in the MATLAB scripts below. We could not get our Interpolate function to work properly, so we got help from Group 2 and the V-cycle worked with their implementation of the interpolation function.

```
function Rc=coarsen(R,m)
    mc = (m-1)/2;
    RR = zeros(m,m);
    for ii = 1:m
        RR(m-(ii-1),:) = R(1+(ii-1)*m:ii*m);
    end
    RR = RR(2:m-1,2:m-1);
%      size(RR)
    Rc = zeros(mc^2,1)+10;
    %hc = 1/(mc+1);
    for ii = 1:mc
        for jj = 1:mc
            Rc(jj+(ii-1)*mc) = RR((m-2)-2*(ii-1),1+2*(jj-1));
        end
    end
end
```

```
function [R,mf] = Interpolate(Rc,mc)

%Construct Coarse Grid Incl. Boundary 0 and 1
hc = 1/(mc+1);
[Xc,Yc] = meshgrid(0:hc:1,0:hc:1);

%Construct Finer Grid where Interpolation is Desired
mf = 2*mc + 1;
hf = 1/(mf + 1);
[Xf,Yf] = meshgrid(0+hf:hf:1-hf,0+hf:hf:1-hf);

%Reshape Residual Values and Apply Zero-Padding
%Remember - Error is Zero at Boundary Points.
Rc_M = rot90(reshape(Rc,mc,mc));
I = zeros(mc+2);
I(2:mc+1,2:mc+1) = Rc_M;

%Interpolate
R = interp2(Xc,Yc,I,Xf,Yf);
R = rot90(reshape(R,mf^2,1));
end
```

Together with the two functions already constructed, *Amult.m* and *smooth.m*, it is possible to execute the Multigrid V-cycle. The cycle works by recursively coarsening the grid and smoothing the error until the error has been mapped down to a single grid point. It is at this point possibly to directly solve the equation that arises and then interpolate back. The continuous process of coarsening until the interior grid points reach $m = 1$ and then re-interpolating back to the original interior points is what contributions to the symbolic 'V' in the name, which indicates a certain "going all the way down and going all the way back up" approach. We have implemented the V-cycle with our scripts and used it in a driver script. We include this script for completeness and for possible debugging.

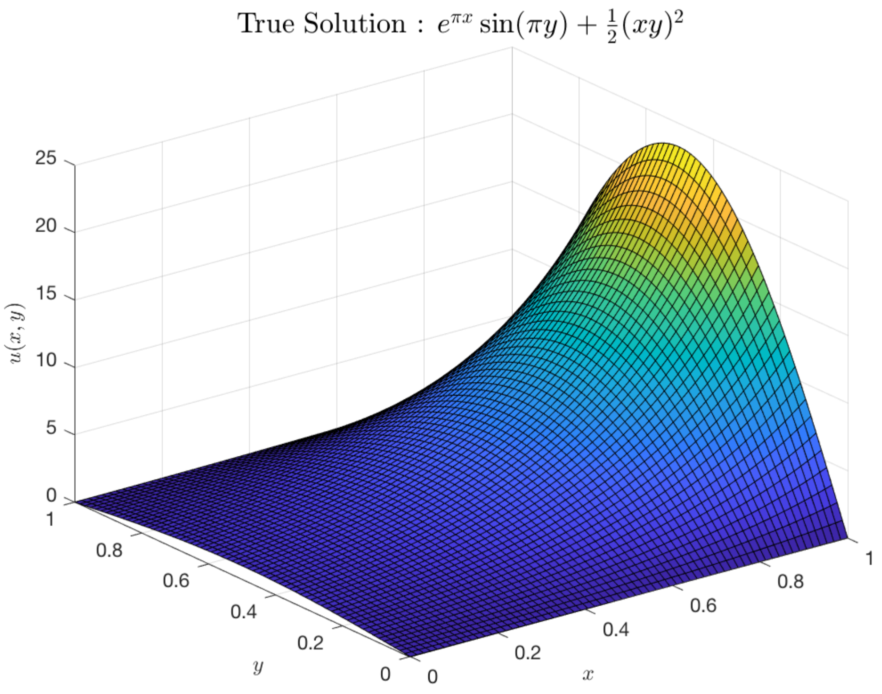
```
function Unew = Vcycle2(U,omega,nsmooth,m,F)
% Approximately solve: A*U = F

l2m = log2(m+1);
assert(l2m==round(l2m));     %m must be m = 2^k - 1 otherwise error
assert(length(U)==m*m);      %u must have length m^2
if (m==1)
    % if we are at the coarsest level
    % TODO: solve the only remaining equation directly!
        Unew = F/16;
else
    % 1. TODO: pre-smooth the error
    %    perform <nsmooth> Jacobi iterations
        for n = 1:nsmooth
            U = smooth2(U,omega,m,F);
        end
    % 2. TODO: calculate the residual
        R = - (Amult(U,m) + F);
    % 3. TODO: coarsen the residual
        Rc = coarsen(R,m);
    % 4. recurse to Vcycle on a coarser grid
        mc = (m-1)/2;
        Ecoarse = Vcycle2(zeros(mc*mc,1),omega,nsmooth,mc,-Rc);
    % 5. TODO: interpolate the error
        %E = Interpolate(Ecoarse,mc);
        E = interpolate_a(Ecoarse,2*mc+1);
    % 6. TODO: update the solution given the interpolated error
        U = U + E;
    % 7. TODO: post-smooth the error
    %    perform <nsmooth> Jacobi iterations
        for n = 1:nsmooth
            Unew = smooth2(U,omega,m,F);
        end
end
end
```

```
clear all; close all; clc

% Exact Solution and RHS
u = @(x,y) exp(pi*x).*sin(pi*y)+0.5*(x.*y).^2;   %True Solution
f = @(x,y) x.^2 + y.^2;                           %Right Hand Side Func
m = 2^6 - 1;                                      %Grid
U = zeros(m*m,1);                                 %Starting Guess
F = form_rhs(f,u,m);                             %Right-Hand-Side

%Plot Graph
h=1/(m+1);
[X,Y] = meshgrid(0:h:1,0:h:1);
figure(2)
surf(X,Y,u(X,Y))
xlabel('$x$','Interpreter','Latex')
ylabel('$y$','Interpreter','Latex')
zlabel('$u(x,y)$','Interpreter','Latex')
set(gca,'Fontsize',11)
title('True Solution : $e^{\pi x} \sin(\pi y) + \frac{1}{2}(xy)^{2}$','Interpreter','Latex','Fontsize', 16
omega = 0.5;
epsilon = 1.0E-6;

for i = 1:1
    R = F + Amult(U,m);
    fprintf('*** Outer iteration: %3d, rel. resid.: %e\n', ...
        i, norm(R,2)/norm(F,2));
    if (norm(R,2)/norm(F,2) < epsilon)
        break;
    end
    U = Vcycle2(U,omega,3,m,F);
    plotU(m,U);
    pause(.5);
end

function plotU(m,U)
figure(1)
h=1/(m+1);
x=linspace(h,1-h,m);
y=linspace(h,1-h,m);
[X,Y]=meshgrid(x,y);
surf(X, Y, reshape(U,[m,m])');
shading interp;
title('Computed solution');
xlabel('x');
ylabel('y');
zlabel('U');
xlabel('$x$','Interpreter','Latex')
ylabel('$y$','Interpreter','Latex')
zlabel('$u(x,y)$','Interpreter','Latex')
set(gca,'Fontsize',11)
title('Iterate Solution : $e^{\pi x} \sin(\pi y) + \frac{1}{2}(xy)^{2}$','Interpreter','Latex','Fontsize',
end
```

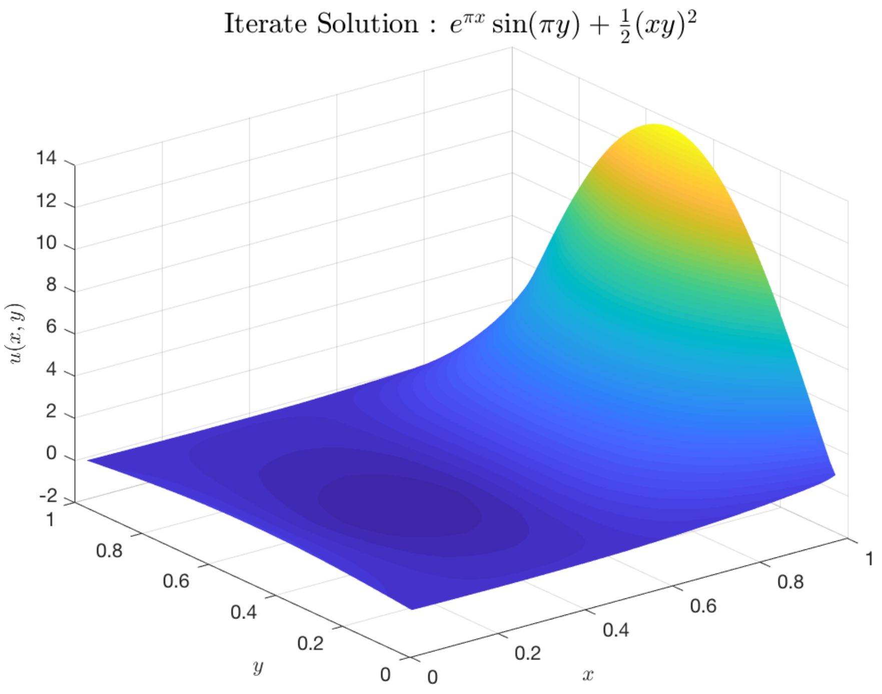
The solution on the trial problem defined with true solution u and right hand side function f

$$u(x,y) = \exp(\pi x)\sin(\pi y)(xy)^2$$
$$f(x,y) = x^2 + y^2$$

is found in Figure (8), where the solution using the Multigrid Algorithm is provided as well.



(a)



(b)

Figure 8: A comparison of the true solution (a) and the solution found by the application of the Multigrid Method, using a V-cycle. (b). $k = 6$, 47 iterations. A tolerance of $\epsilon = 10^{-6}$

For a given tolerance, we can plot the the number of iterations needed to reach the tolerance, as a function of the grid size. The results show faster convergence for larger grid size. This is not what we expect. In the 1D case presented in the lectures, the number of iterations was constant as a function of grid size. The plot is shown in Figure (9). We used a tolerance of $\epsilon = 10^{-6}$. It is however worth mentioning that comparison with other groups revealed that our method required more iterations, which could indicate some small issues. As already mentioned previously the attempt to construct certain functions according to naturally ordering seemed to cause issues sometimes as the vector/matrix elements are shuffled indesirably.

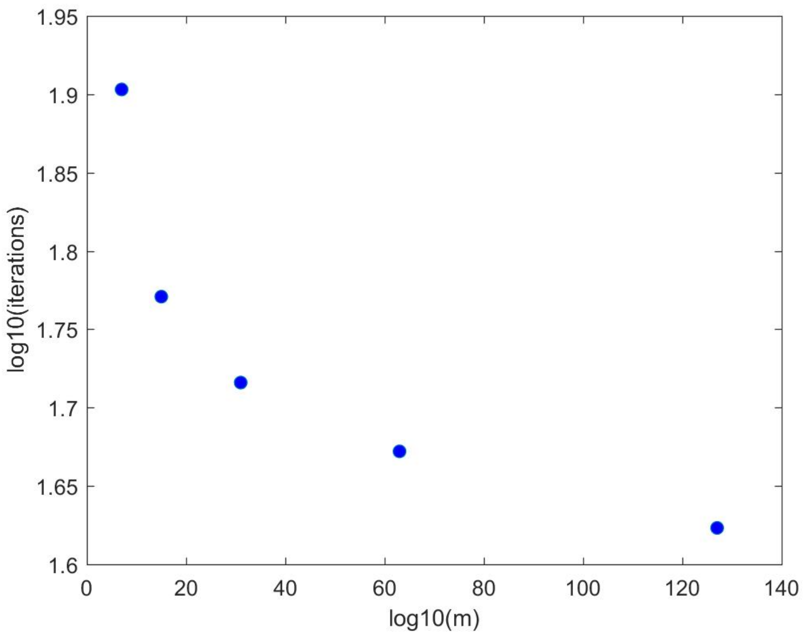


Figure 9: The iterations to reach tolerance $\epsilon = 10^{-6}$ versus m on logarithmic axes.

In conclusion the implementation does not seem to be satisfactory however some of the remaining smaller issues have been highlighted through the text. It is however deemed slightly successful nonetheless as convergence was acquired and the true solution approximated quite well.

A MATLAB code

A.1 G(U) evaluation: GEval.m

```
function G = GEval(u,m,h,alpha,beta,epsilon)
    % Apply equation (2.106) in the book
    % First point (uses the left boundary condition)
    G(1) = epsilon*(alpha-2*u(1)+u(2))/h^2 + u(1)*((u(2)-alpha)/(2*h)-1);
    for ii = 2:m-1 % All interior points
        G(ii) = epsilon*(u(ii-1)-2*u(ii)+u(ii+1))/h^2 + u(ii)*((u(ii+1)-u(ii-1))/(2*h)-1);
    end
    % Last point (uses the right boundary condition)
    G(m) = epsilon*(u(m-1)-2*u(m)+beta)/h^2 + u(m)*((beta-u(m-1))/(2*h)-1);
    G=G';
end
```

A.2 Jacobian evaluation: JacEval.m

```
function J = JacEval(U,n,h,alpha,beta,epsilon) % Evaluate the Jacobian
    % First row (uses left boundary condition)
    J(1,1) = -2*epsilon/h^2 + (U(2)-alpha)/(2*h)-1;
    J(1,2) = epsilon/h^2 + U(1)/(2*h);
    for ii = 2:n-1 % All interior point
        J(ii,ii-1) = epsilon/h^2 - U(ii)/(2*h);
        J(ii,ii) = -2*epsilon/h^2 + (U(ii+1)-U(ii-1))/(2*h)-1;
        J(ii,ii+1) = epsilon/h^2 + U(ii)/(2*h);
    end
    % Last row (uses the right boundary condition)
    J(n,n-1) = epsilon/h^2 - U(n)/(2*h);
    J(n,n) = -2*epsilon/h^2 + (beta-U(n-1))/(2*h)-1;
end
```

A.3 IVP and Sensitivities

```
function  zdot = IVP(t,z,epsilon)
%Function that computes the derivative of x and of the sensitivities with
%respect to the starting condition x0

%Unpack Variables
x = z(1:2);
sx = z(3:end);
Sx = reshape(sx,2,2);

%Computing xdot
x1 = x(1);
x2 = x(2);
xdot(1) = x2;
xdot(2) = (x1*(1-x2))/epsilon;
xdot = xdot';

%Computing Sxdot
F = [0 , 1 ; (1-x2)/epsilon , x1/epsilon];
Sxdot = F*Sx;

zdot = [xdot ; Sxdot(:)];
end
```

References

[LeVeque, 2007] LeVeque, R. J. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations.* Society for Industrial and Applied Mathematics.