Technical University of Denmark

02685 Scientific Computing for Differential Equations
2017

Assignment 2

Authors:
Miguel Suau de Castro (s161333)
Michal Baumgartner (s161636)
March 16, 2017

Contents

1 2 point Boundary Value Problems

2-point Boundary Value Problems (BVP) represent a special case of differential equations where the solution is limited by a pair of constraints at both ends of the interval. Hence, the task consists of finding the solution that apart from solving the differential equation satisfies the boundary conditions.

In the linear case, a numerical approximation to the solution to these problems can be obtained solving a linear system. However, for nonlinear differential equations the solution is not straightforward and an iterative process is needed. In this exercise, we shall study different methods for solving nonlinear BVPs by applying them to the following differential equation:

$$\begin{aligned} \epsilon u(t)'' + u(t)(u(t)' - 1) = 0 \qquad & 0 \leq t \leq 1 \\ u(0) = \alpha \qquad & u'(1) = \beta \end{aligned} \tag{1}$$

Note that for convenience the report has been structured differently than in the assignment specifications and the answer to some of the questions may be located in other sections.

1.1 Newton's method

If the independent variable t is discretized and the second derivative in equation 1 is replaced by the centered difference method, an approximation to the solution at a series of equidistant points can be found solving the following scheme:

$$\begin{aligned} \epsilon \left(\frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} \right) + u_i \left(\frac{u_{i+1} - u_{i-1}}{2h} - 1 \right) = 0 \\ u(0) = \alpha \qquad u(1) = \beta \end{aligned} \tag{2}$$

Where $i = 1, 2, ..., m$ and h is the distance between points in the interval.

The centered approximation comes from a Taylor series expansion truncated to the second term. We can show that the approximation is second-order accurate by computing the local error:

$$\begin{aligned} \tau_i = \epsilon \left(\frac{u(t_{i-1}) - 2u(t_i) + u(t_{i+1})}{h^2} \right) + u_i \left(\frac{u(t_{i+1}) - u(t_{i-1})}{2h} - 1 \right) - \\ \epsilon u(t_i)'' - u(t_i)(u(t_i)' - 1) \end{aligned} \tag{3}$$

The above expression is just the difference between the given differential equation and its finite difference approximation. By applying the Taylor series expansion to the first two terms:

$$\tau_i = \epsilon\left(u''(t_i) + \frac{1}{12}h^2u'''(t_i) + O(h^4)\right) + u(t_i)\left(u'(t_i) + \frac{1}{6}h^2u'''(t_i) + O(h^5) - 1\right) - \epsilon u(t_i)'' - u(t_i)(u(t_i)' - 1) \quad (4)$$

$$\tau_i = \epsilon\left(\frac{1}{12}h^2u'''(t_i) + O(h^4)\right) + u(t_i)\left(\frac{1}{6}h^2u'''(t_i) + O(h^5)\right) \quad (5)$$

The dominant term in equation 5 is then dependent of h^2 and thus we can conclude that the local truncation error is order $O(h^2)$.

Equation 6 can be expressed as a nonlinear system of the form:

$$G(U) = 0 \quad (6)$$

And its roots can be obtained using Newton's method.

$$U^{[k+1]} = U^{[k]} + J(U^{[k]})^{-1}G(U^{[k]}) \quad (7)$$

Where $U^{[k]}$ is a vector containing a discrete approximation to the solution after k iterations and $J(U^{[k]})$ is the Jacobian matrix of the system, which in our case:

$$J(U) = \begin{cases} \frac{\epsilon}{h^2} - \frac{u_i}{2h} & j = i-1 \\ -\frac{2\epsilon}{h^2} - \frac{u_{i+1}-u_{i-1}}{2h} - 1 & j = i \\ \frac{\epsilon}{h^2} + \frac{u_i}{2h} & j = i-1 \end{cases} \quad (8)$$

Our implementation of the Newton's method consist of two functions: FunJac, which builds the vector G and the tridiagonal matrix J and NewtonsMethod, which updates the solution applying 7 until the tolerance is satisfied or the number of iterations is greater than the threshold.

The convergence and accuracy of the method depend on the proximity of the initial guess to the real solution. An approximation to the solution is given by equation 2.105 in LeVeque. Since an arbitrary choice might lead to non-convergence we have used this approximation as the initial value. Figure 1 shows the initial guess along with the solution returned by the Newton's method for 50 grid points.

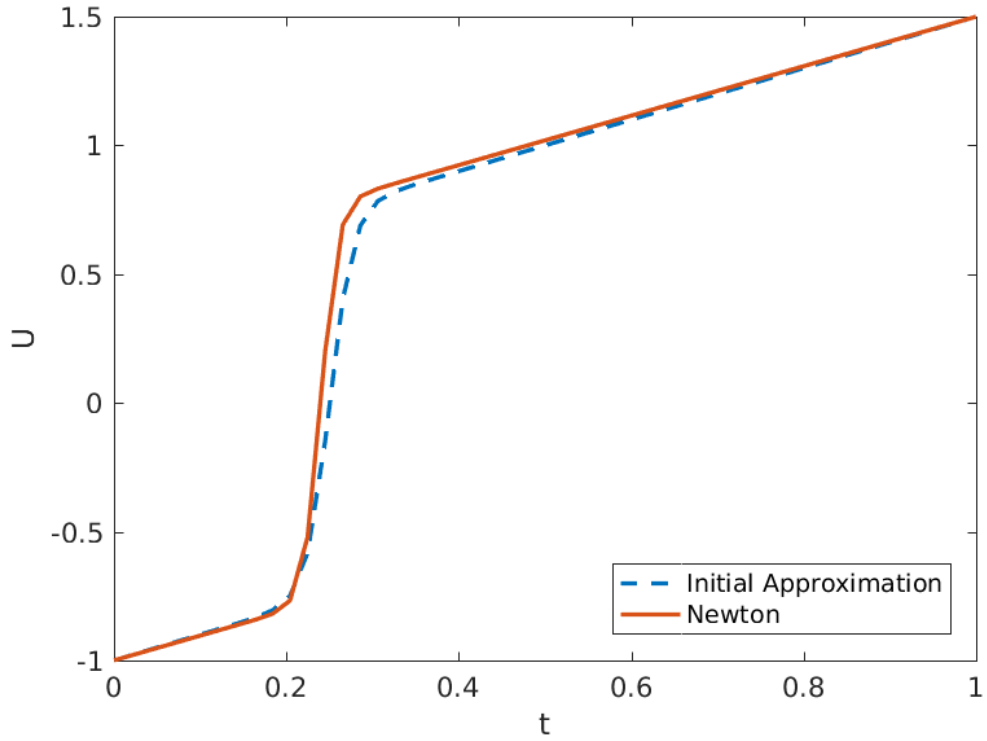

Figure 1: Initial approximation and solution to the BVP using Newton's method for 50 grid points

The solution to 1 has also been computed using bvp4c function in MATLAB. Figure 2 shows a comparative of the results obtained using Newton's method and bvp4c setting the tolerance equal to 0.01 and 33 grid points.

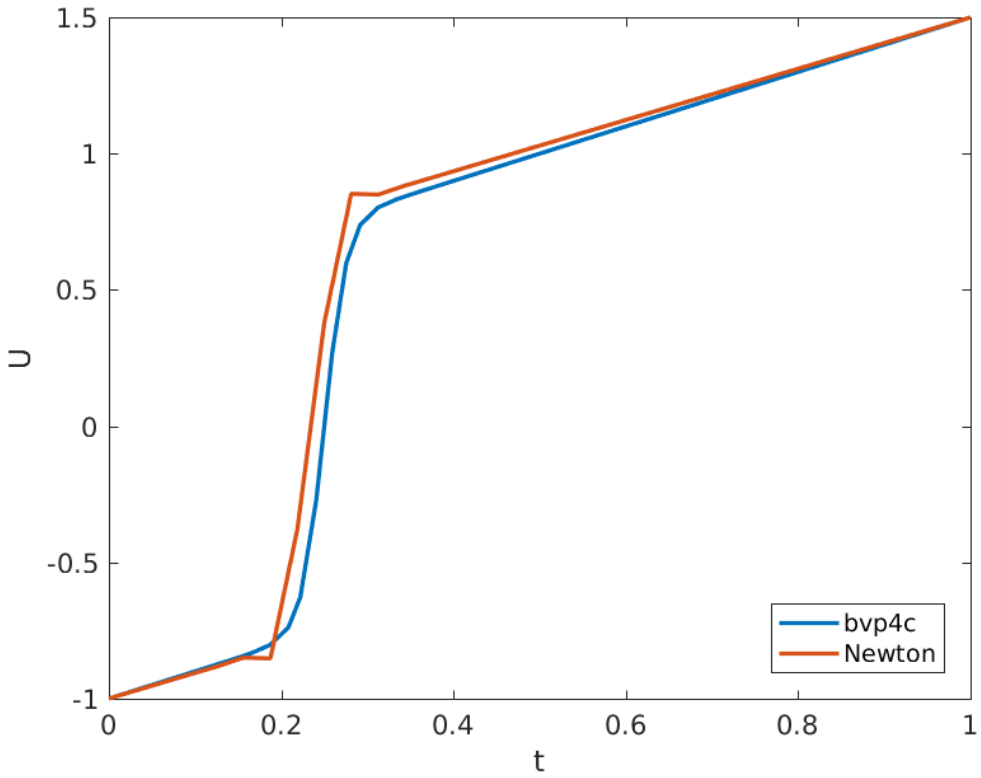

Figure 2: Initial approximation and solution to the BVP using Newton's method and bvp4c with 33 grid points

Finally a more accurate approximation has been obtained setting the tolerance of bvp4c to 10^{-10}. A section of the solution is shown in figure 3 along with the Newton's method solution now for 100 grid points. The region shown in the graph corresponds to a layer where the solution changes rapidly. It is easy to see that Newton's method lies further from the true solution in this region. The number of points required to obtain a more accurate solution is greater than anywhere else due to this sudden variation. Hence, better results could be obtained using a non-uniform grid. The solution would be more finely discretized in the mentioned region and thus any rapid change could be easily controlled.

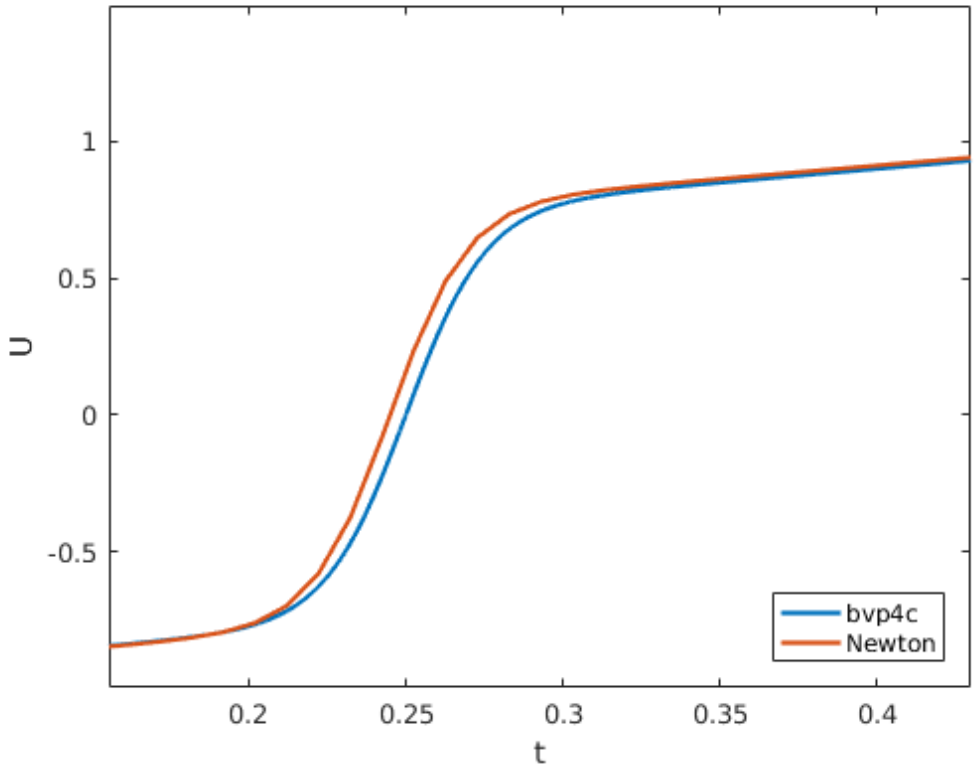

Figure 3: Section of the solution to the BVP using Newton's method with 100 grid points and bvp4c for a tolerance of 10^{-10}

1.2 Single shooting

One could also write the BVP in equation 1 as an initial value problem (IVP) of the form:

$$\begin{aligned} u''(t) = \frac{1}{\epsilon}u(t)(u'(t)-1) \qquad & 0 \leq t \leq 1 \\ u(0) = \alpha \qquad u'(0) = \sigma & \end{aligned} \tag{9}$$

and splitting $u(t)$ and $u'(t)$ into two different variables $u_1(t)$ and $u_2(t)$ respectively:

$$\begin{aligned} u_2'(t) = \frac{1}{\epsilon}u_1(t)(u_2(t)-1) & \\ u_1'(t) = u_2(t) & \\ u_1(0) = \alpha \qquad u_2(0) = \sigma & \end{aligned} \tag{10}$$

Note that a new variable σ has been included whose value must be chosen so that the second constraint in 1 is satisfied:

$$r(\sigma) = U_1(1, \sigma) - \beta = 0 \tag{11}$$

An approximation to the solution of the IVP can be computed in MATLAB using the function ode45. The naive way to tackle the problem would be to try different values of σ and choose the one that yields the closest solution to the boundary constraint. However, the use of a numerical algorithm to find the root of 11 would be computationally more convenient.

As a first approach we implemented the bisection method. Given two values of σ for which equation 11 has opposite sign we know that the solution will lie within the interval described by these two points. Consequently the solution is found by progressively reducing the interval until its size is below a tolerance. Although the algorithm will converge no matter the size of the interval, the problem now is translated into obtaining the two initial points. Moreover, the method presents a poor linear rate of convergence.

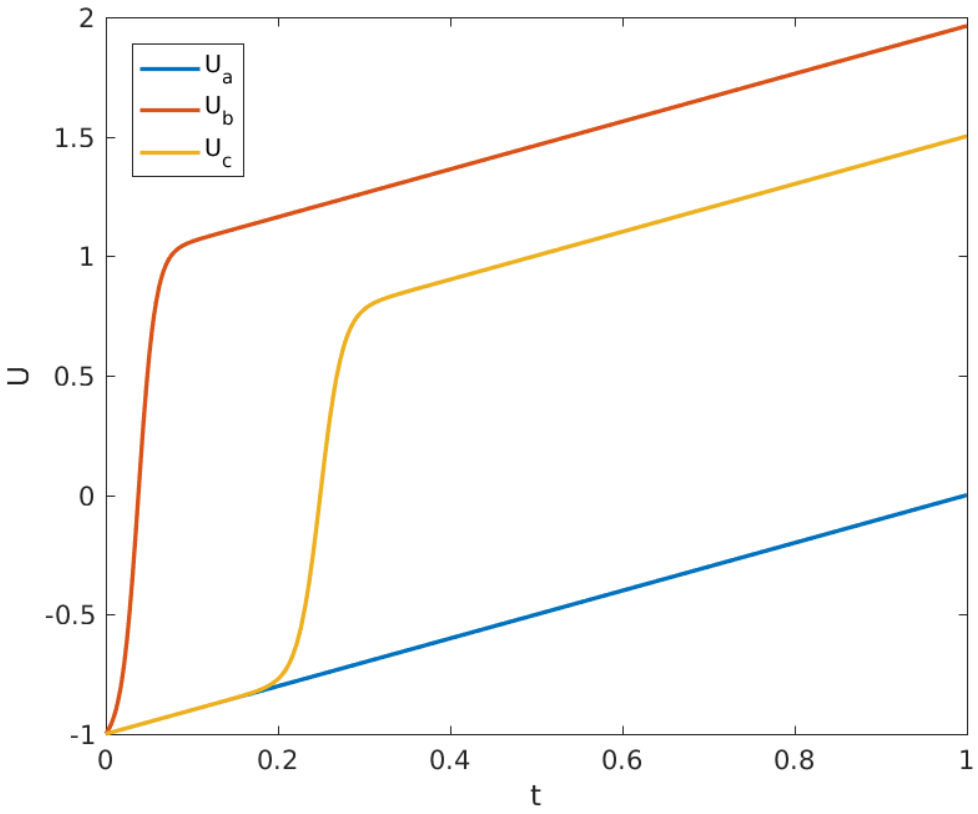

Figure 4: Solution to the IVP using the bisection method. U_a and U_b are the two initial guess of opposite sign while U_c is the solution that verifies the second boundary constraint

Figure 4 shows the solution obtained U_c applying the bisection method along with the two initial guesses U_a and U_b, of opposite sign with respect to the boundary constraint. We see that after a certain number of iterations the algorithm is able to hit the second constraint.

A more efficient algorithm can be obtained applying Newton's method. For that, we would like to have an expression for the first partial derivative of 11 with respect to σ so that the value of σ can be updated in every iteration.

Since β in 11 is a constant we can write:

$$\frac{\partial r(\sigma)}{\partial \sigma} = \frac{\partial u(1,\sigma)}{\partial \sigma} \tag{12}$$

taking the partial derivative with respect to σ in 10:

$$\frac{\partial u''}{\partial \sigma} = \frac{-u' + 1}{\epsilon}\frac{\partial u}{\partial \sigma} - \frac{u}{\sigma}\frac{\partial u'}{\partial \sigma}$$
$$\frac{\partial u(0)}{\partial \sigma} = 0 \qquad \frac{\partial u'(0)}{\partial \sigma} = 1 \tag{13}$$

and since this is also an IVP it can be solved together with 10 as a single differential equation. As we mentioned before, Newton's method requires a suitable initial guess to converge. We experienced in this case that the algorithm is unable to converge for small values of ϵ in such cases, the bisection method should be applied. Figure 5 shows the solution at the end of the interval for a series of σ values. Although a more detailed reason for the non-convergence of the method is given in the next section, it easy to see that the function is close to be non-differentiable in a neighborhood of σ optima.

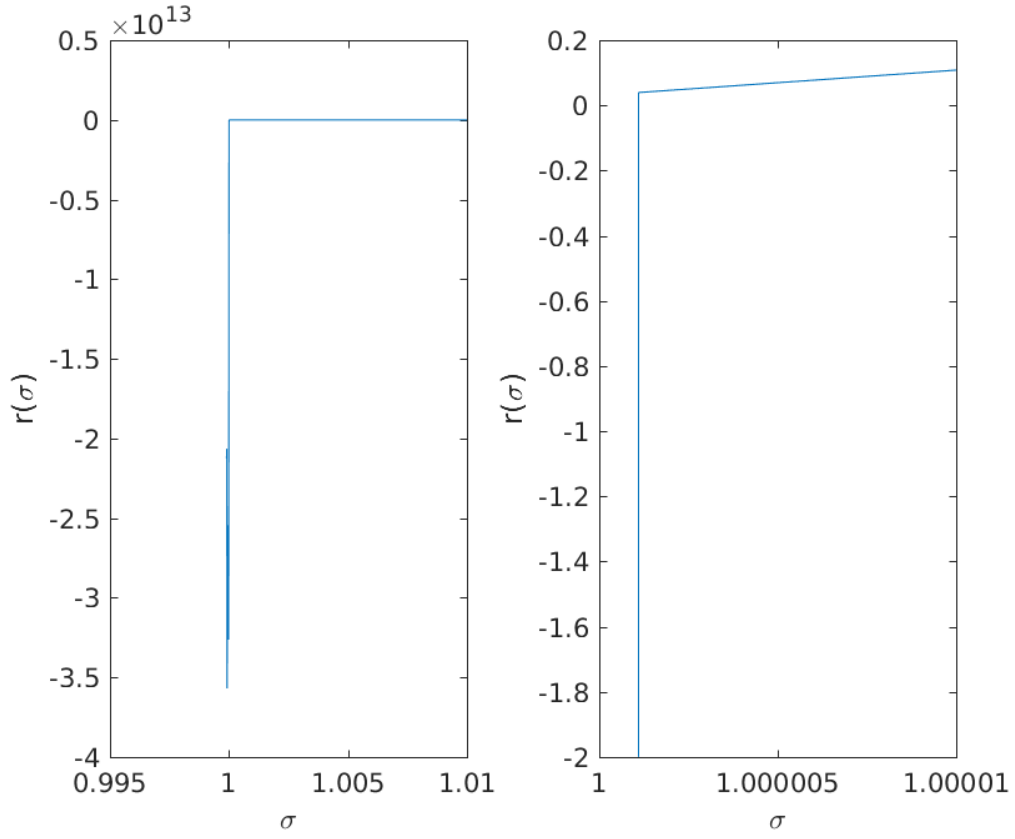

Figure 5: Solution to the IVP at the last point of the interval for a series of σ values close to the optima with $\epsilon = 0.01$

The rate of convergence of both algorithms has been plotted and can be seen in

figure 6. The graph shows that for ϵ sufficiently large Newton's method converges quadratically.

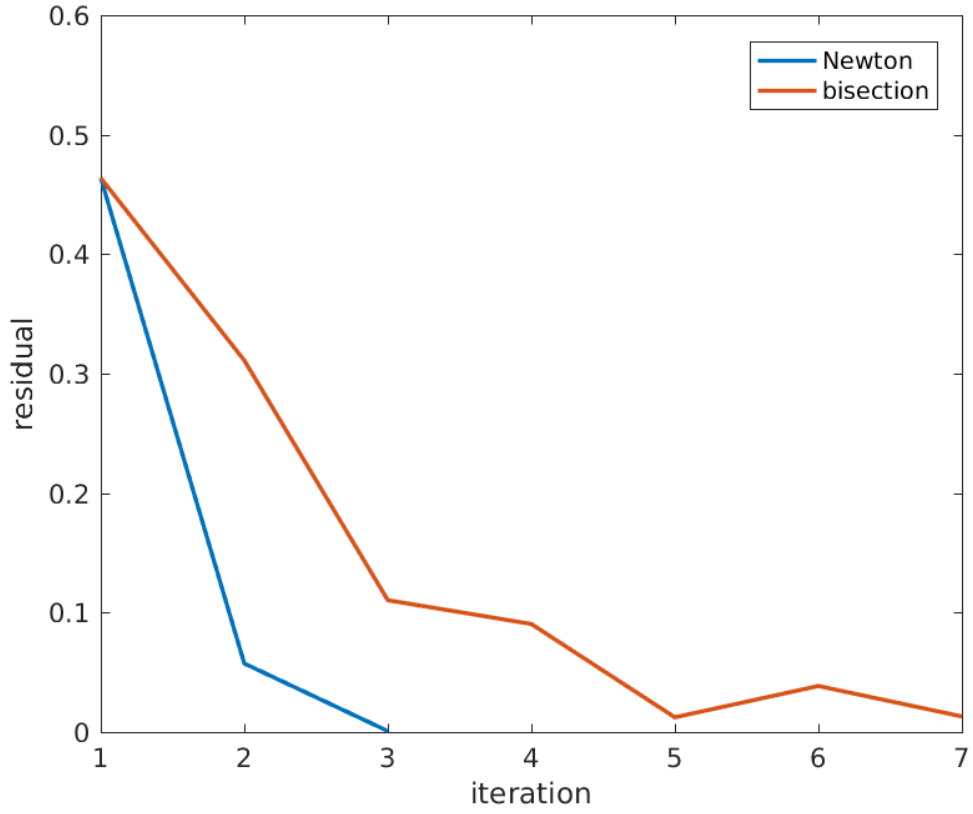

Figure 6: Newton and bisection algorithm rate of convergence for $\epsilon = 0.8$

Finally the secant method is a variation of the Newton's algorithm where the derivative is approximated by finite difference so that there is no need for solving an extra differential equation. The code for all three algorithms is collected in the appendix. Besides, some information about the three methods performance is shown in table 1.

1.3 Sensitivity analysis and non-convergence

In order to find an approximation to the partial derivative of the solution with respect to σ we can either solve equation 13 or apply the finite difference method. In both cases we see that for small values of ϵ the solution becomes very sensible to small variations of σ (table 2). On the other hand, as we saw in figure 5

Method	Iterations	CPU-time
Bisection	8	0.0054
Newton	3	0.0022
Secant	3	0.0036

Table 1: Number of iterations and CPU-time of all three algorithms with $\epsilon = 0.8$

ϵ	S_σ
0.01	1.0299×10^5
0.05	0.5791
0.1	0.3620
0.5	0.7110

Table 2: Sensitivity of the solution with respect to σ at the final point for different values of ϵ

equation 11 was close to be non-smooth (non-differentiable) in a neighborhood of the solution. That is the reason why neither Newton's method nor the secant algorithm are able to converge even for initial guesses close to the true solution.

If the initial value lies where the slope in figure 5 is almost 0 the method takes a very large step in the first update and the algorithm is unable to find the way back. If on the other hand, the initial value fells in a region where the curve is very steep, the step size in every iteration will be very small and the algorithm will never converge.

A way around would be to include an extra parameter that controls the step size every time σ is updated. However, in this case the number of iterations increases dramatically for small values of ϵ. Thus, it might be advisable to use an algorithm that does not require partial derivatives, such as the bisection method. Applying the bisection method with $\epsilon = 0.01$ the algorithm converges after 24 iterations.

2 9-point Laplacian

This section will extend the implementation of the 5-point Laplacian to a 9-point stencil and discuss how to form the right hand side of the Poisson equation using the correction 3.19 from LeVeque to achieve $\mathcal{O}(h^4)$ convergence in terms of global error (methods to find it can be found in the appendix of LeVeque).

The spacial domain will be unit square defined as $[0, 1] \times [0, 1]$ and discretized to a regular grid with $m \times m$ interior points ($(m + 2) \times (m + 2)$ with bound-

aries). Dirichlet BC are prescribed everywhere on the boundary, but the method `form_rhs(m,f,u)` will use the "exact" u to calculate them for the sake of simplicity.

2.1 9-point stencil

Visualization of the stencil is shown in figure 3.1b of LeVeque and is achieved in a similar manner as the 5-point stencil, that is by replacing the derivatives with centered finite differences to get the discretization. Since the grid is uniform this leads to equation 3.10 (LV) for the 5-point stencil:

$$\frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) = f_{i,j} = \nabla_5^2 u_{i,j} \tag{14}$$

It can be seen that a "cross" like shape is used with the coefficients adding up to 0. As mentioned in the book, this stencil has some drawbacks so let's look at the 9-point one.

The approximation is given by

$$\begin{aligned}\nabla_9^2 u_{i,j} = \frac{1}{6h^2}(&4u_{i-1,j} + 4u_{i+1,j} + 4u_{i,j-1} + 4u_{i,j+1} \\ &+ u_{i-1,j-1} + u_{i-1,j+1} + u_{i+1,j-1} + u_{i+1,j+1} \\ &- 20u_{i,j})\end{aligned} \tag{15}$$

in order to achieve the fourth-order accuracy a clever expansion is presented in LeVeque: the approximation is applied to the true solution and expanded into a Taylor series, with the dominant error term rewritten as $u_{xxxx} + 2u_{xxyy} + u_{yyyy} = \nabla^2(\nabla^2 u) \equiv \nabla^4 u$. For Poisson equation $\nabla^2 u = f$, it is possible to further rewrite $u_{xxxx} + 2u_{xxyy} + u_{yyyy}$ as $\nabla^2 f$. Given these results, the dominant error term can be computed with just the knowledge of the function f. More so, if f is *zero* or a *harmonic* function, then the aforemention term in the local truncation error disappears and the 9-point Laplacian yields a fourth order accurate discretization.

This approach can be extended for arbitary smooth functions f by defining

$$\nabla_9^2 u_{ij} = f(x_i, y_j) + \frac{h^2}{12}\nabla^2 f(x_i, y_j) \tag{16}$$

to cancel out the $\mathcal{O}(h^2)$ terms and end up with $\mathcal{O}(h^4)$ as the local truncation error of this method.

Particularly useful trick for numerical methods when the values of f are only known at the grid points is by using the 5-point Laplacian as follows:

$$\nabla_9^2 u_{ij} = f(x_i, y_j) + \frac{h^2}{12}\nabla_5^2 f(x_i, y_j). \tag{17}$$

Having $\mathcal{O}(h^4)$ out of the way, it is possible to implement the `form_rhs` method by using `poisson9` and `poisson5` function to calculate the respective Laplacians.

2.2 Case problems

Three equations were given to test out the implementation. The first equation is used to make sure that the desired convergence in terms of the global error is achieved.

$$u_{0,exact}(x,y) = \sin(4\pi(x+y)) + \cos(4\pi xy) \tag{18}$$
$$u_{1,exact}(x,y) = x^2 + y^2 \tag{19}$$
$$u_{2,exact}(x,y) = \sin(2\pi|x-y|^{2.5}) \tag{20}$$

Functions u_1 and u_2 will have different rates of convergence of the global error, explanation as to why will be provided later on in this section.

For all the aforementioned functions u, functions f were simply found analytically. One can use Mathematica, Matlab's Symbolic Toolbox or any other CAS to obtain them.

First case

$$u_0(x,y) = \sin(4\pi(x+y)) + \cos(4\pi xy) \tag{21}$$
$$f_0(x,y) = -16\pi^2\left(\cos(4\pi xy)x^2 + \cos(4\pi xy)y^2 + 2\sin(4\pi(x+y))\right) \tag{22}$$

The method `form_rhs(m,f,u)` accepts m as the dimension of the inner point grid $m \times m$, f is the function handle for the RHS of $\nabla^2 u = f$ and finally u is passed in as a function handle to calculate the boundaries of the grid. Note that u is passed in only for simplicity, one could add in an option to provide the boundaries as a matrix instead.

Solution is simply found by using the direct method of solving a system $Au = f$ using Matlab's backslash operator.

In this case the function f_0 is harmonical and according to LeVeque, using a 9-point Laplacian with correction we are able to achieve fourth order convergence.

Second case

$$u_1(x,y) = x^2 + y^2 \tag{23}$$
$$f_1(x,y) = 4 \tag{24}$$

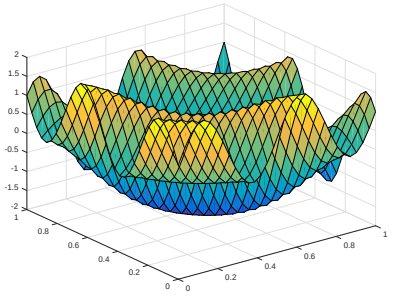

Figure 7: Calculated u_0 using `poisson9 \form_rhs` with BC on 32×32 grid

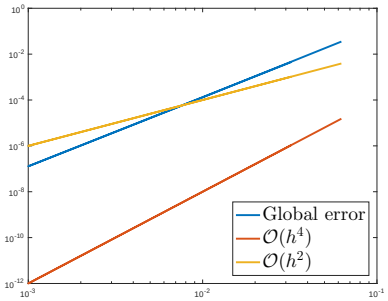

Figure 8: Global error (blue), $\mathcal{O}(h^4)$ (orange), $\mathcal{O}(h^2)$ (yellow)

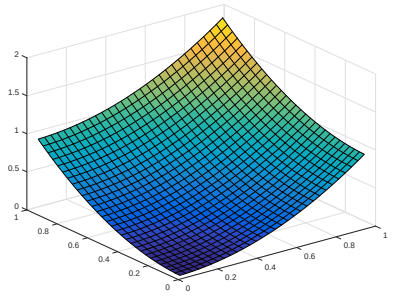

Figure 9: Calculated u_1 using `poisson9 \form_rhs` with BC on 32×32 grid

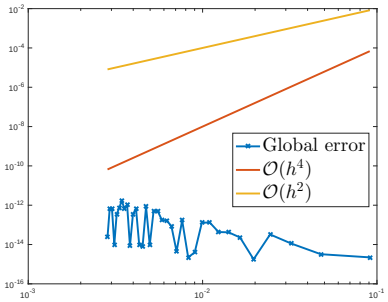

Figure 10: Global error (blue), $\mathcal{O}(h^4)$ (orange), $\mathcal{O}(h^2)$ (yellow)

Function f_1 is a constant and the global error osciallates. This is due to the fact that the machine precision plays a role in over or undershooting the exact solution resulting in oscillations. Note that the markers indiciating the respective h were added for clarity in the global error (loglog) plot.

Third case

$$u_2(x,y) = \sin(2\pi|x-y|^{2.5}) \tag{25}$$

$$f_2(x,y) = 5\pi\sqrt{|x-y|}\,\mathrm{sgn}(x-y)^2\left(3\cos(2\pi|x-y|^{\frac{5}{2}}) - 10\pi|x-y|^{\frac{5}{2}}\sin(2\pi|x-y|^{\frac{5}{2}})\right) \tag{26}$$

f_2 has a singularities, which can be checked by finding the exact form of the Laplacian in Mathematica for example – the result contains first and second

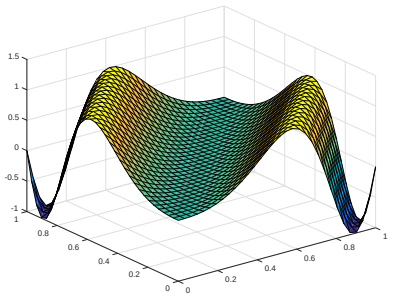

Figure 11: Calculated u_2 using `poisson9 \form_rhs` with BC on 32×32 grid

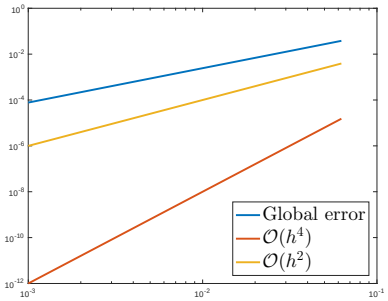

Figure 12: Global error (blue), $\mathcal{O}(h^4)$ (orange), $\mathcal{O}(h^2)$ (yellow)

derivatives of $x - y$, which are undefined when $x = y$. Still, something close to the second order accuracy is obtained.

3 Iterative solvers in 2D

This section goes through necessary parts needed to build a V-cycle solver on a 2D grid with the same assumptions as in section 2. Without further ado, let's take a look at a pseudoalgorithm first and then explain what individual functions do in detail. The name V-cycle comes from the recursive part of the algorithm – the 2

```
1  function Vcycle(U, ω, n_smooth, m, F)
2  if m = 1 then
3  |   directly solve at the coarsest level
4  else
5  |   pre-smoothing: smooth
6  |   compute the residual: r = F − Au
7  |   coarsen residuals: coarsen
8  |   recursive part to obtain error: Vcycle(0, ω, n_smooth, −R_coarse)
9  |   interpolate the error: interpolate
10 |   update the solution: U = U + E
11 |   post-smoothing: smooth
12 end
13 return U
```

Algorithm 1: V-cycle algorithm pseudocode, full reference can be found on slide 8 of Lecture 20 or in LeVeque 4.6.2 "The multigrid approach"

lines indicate the path of pre- and post-smoothing on coarse/interpolated grids, whereas the point of the intersection indicates directly finding the solution in the first `if` branch.

It is worth noting that due to coarsened grid the depth of the recursion will be $\log_2(m)$ per one cycle. The cycle will be repeated until a desired tolerance is reach or the maximum number of iterations has passed.

3.1 Matrix free 5-point Laplacian

As requested in the handout, matrix-free method `Amult` will be used to calculate the term $-\mathbf{Au}$ without storing the $\mathbf{A}$ matrix in memory (as it is of size $m^2 \times m^2$ and somewhat sparse). The implementation can be tested with Matlab's Conjugate Gradient algorithm `pcg`, which can accept a function handle as the first argument, so `Amult` shall be provided. The reason why $-\mathbf{Au}$ is used instead of the positive alternative is that PCG requires a positive definite system and the $\mathbf{A}$ matrix is in fact negative definite. To check the difference between the implementation of the

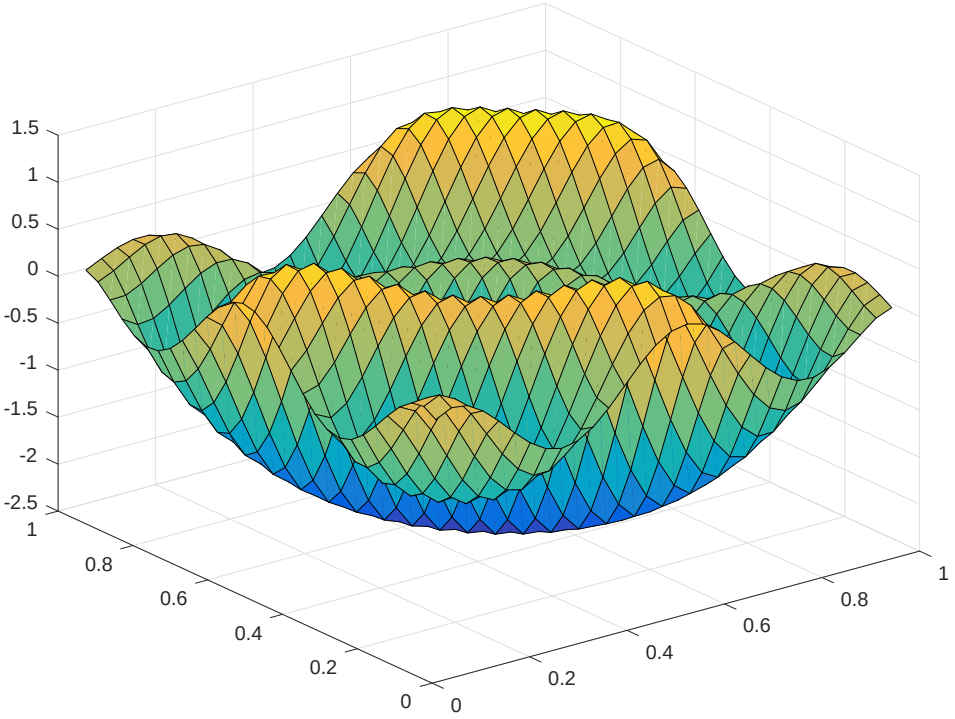

Figure 13: Solution to $-\mathbf{Au} = -\mathbf{f}$ using PCG

matrix free method and the direct multiplication of u with the Laplacian matrix, one can for example use `max(max(abs(Amult(u,m) - -1*poisson5(m)*u)))` in Matlab. The maximum of the absolute difference is roughly 10^{-12}.

All the Matlab code for this section is provided in the appendix.

3.2 Relaxation and smoothing

The main motivation to use underelaxed Jacobi is that while the high frequency components of the error decay after a few iterations when applying the ordinary Jacobi method, the lower frequency components remain longer and negatively affect the convergence of the algorithm.

Thereby, by including an extra parameter ω we can manage how far the current iterate moves towards the true solution, and thus control high frequency harmonics.

$$G_\omega = (1-\omega)I + \omega G \tag{27}$$

where G

$$G = I + \frac{h^2}{2}A \tag{28}$$

besides, considering that the eigenvectors of G are the same as the eigenvectors of A

$$\rho_{p,q} = 2 + \frac{h^2}{2}\lambda_{p,q} \tag{29}$$

from 3.15 in Leveque we know that the eigenvalues of A are

$$\lambda_{p,q} = \frac{2}{h^2}((cos(p\pi h) - 1) + (cos(q\pi h) - 1)) \tag{30}$$

thus, combining the equations above we obtain the following expression for the eigenvalues of G_ω

$$\gamma_{p,q} = (1-\omega) + \omega(cos(p\pi h) + cos(q\pi h)) \tag{31}$$

and since we would like to choose ω so that we get optimal smoothing of high frequencies

$$\omega_{opt} = min(max_{m/2 \leq p,g \leq m}|\gamma_{p,q}|) \tag{32}$$

Figure 14 shows $max_{m/2 \leq p,g \leq m}|\gamma_{p,q}|$ for different values of the smoothing factor. We see in this case that ω_{opt} lies very close to 0.5.

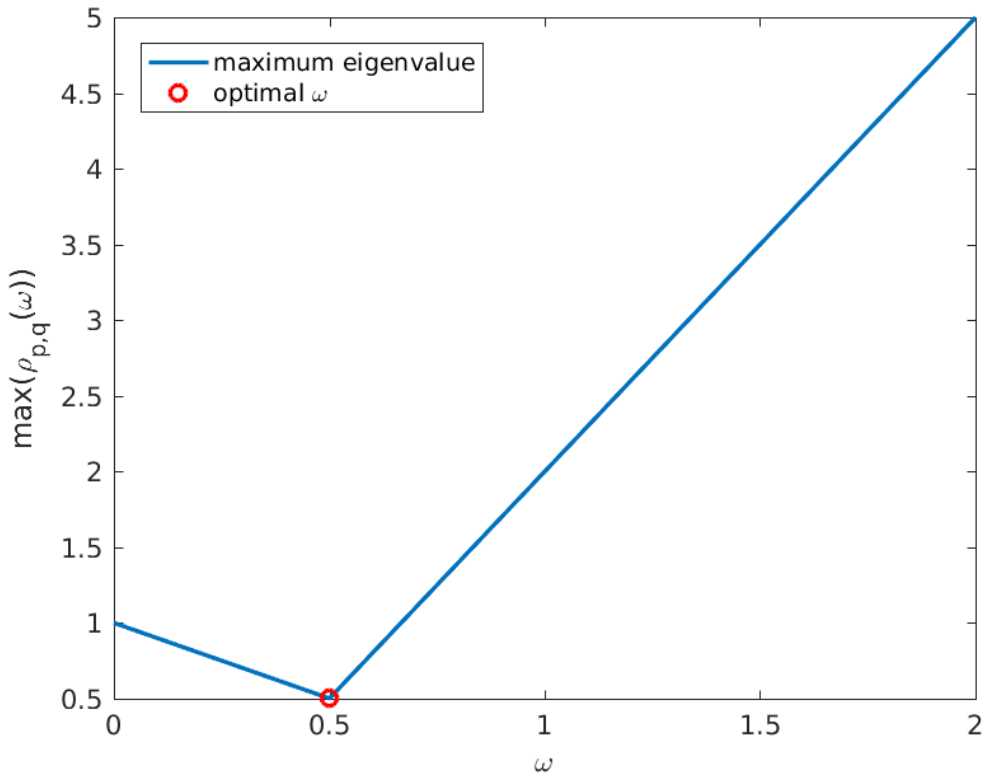

Figure 14: $max_{m/2 \leq p,g \leq m} |\gamma_{p,q}|$ for different values of ω and $m = 1000$

We have implemented a script in MATLAB that plots $max_{m/2 \leq p,g \leq m} |\gamma_{p,q}|$ as a function of ω and finds the minimum of that function. We also wrote the function smooth which applies the under/overrelaxed Jacobi method for a given value of the smoothing parameter.

3.3 Coarsing / interpolating

To further continue, a necessary part of the V-cycle is coarsing and interpolation of the grid. As is mentioned in the book, the convergence rate for some components of the error is greatly improved by transferring the error to a coarser grid. An example is given that a component's frequencies can be shifted to the right, i.e. more in the middle of what range the coarse grid has to offer, resulting in the improvement of the damping factor (in some cases as high as one order of the magnitude).

There are multiple approaches to coarsion and interpolation on an evently spaced squared grid. The simplest one is taking every second point of the original

grid in coarsion and calculating back the points of interpolated grid as a weighted average of the neighbouring points. In this approach three possible scenarios can happen to calculate the interpolated point: adding 2 neighbours vertically, adding 2 neighbours horizontaly and finally, adding 4 neighbours located in NE, NW, SE, SW.

3.4 Recursion

Now the problem is refolmulated in order to transfer the remaining part to the coarser grid. The approach taken uses the recursive call to find the error vector by solving $\mathbf{Ae} = -\mathbf{r}$ via the `vcycle` method.

Since the result is obtained on a coarse grid, interpolating is used in order to project the error vector back to the original grid. In order to find a better approximation of U the error vector is added and post-smoothing is performed. Once all the recursive results are popped from the stack a solution to the given problem is returned. As mentioned before this process is repeated until a desired tolerance or maximum number of iterations is reached.

3.5 Conclusion

With the initial parameters for the V-cycle driver and number of smoothing iterations set to 1000, 23 iterations are needed to reach the desired tolerance of 1^{-10}. Increasing the aforementioned number of smoothing interations results in longer computational time, but decreases the overall number of steps needed for convergence. As can one expect lowering k in $m = 2^k - 1$ yields considerably less iterations, for example for $k = 5$, only 5 run throughs of the for-loop are executed. Having a larger grid would naturally increase the computation time.

Further work can be done to implement the W-cycle or other ways of coarsion and interpolation, such as using a stencil like shape to weigh the grid points.

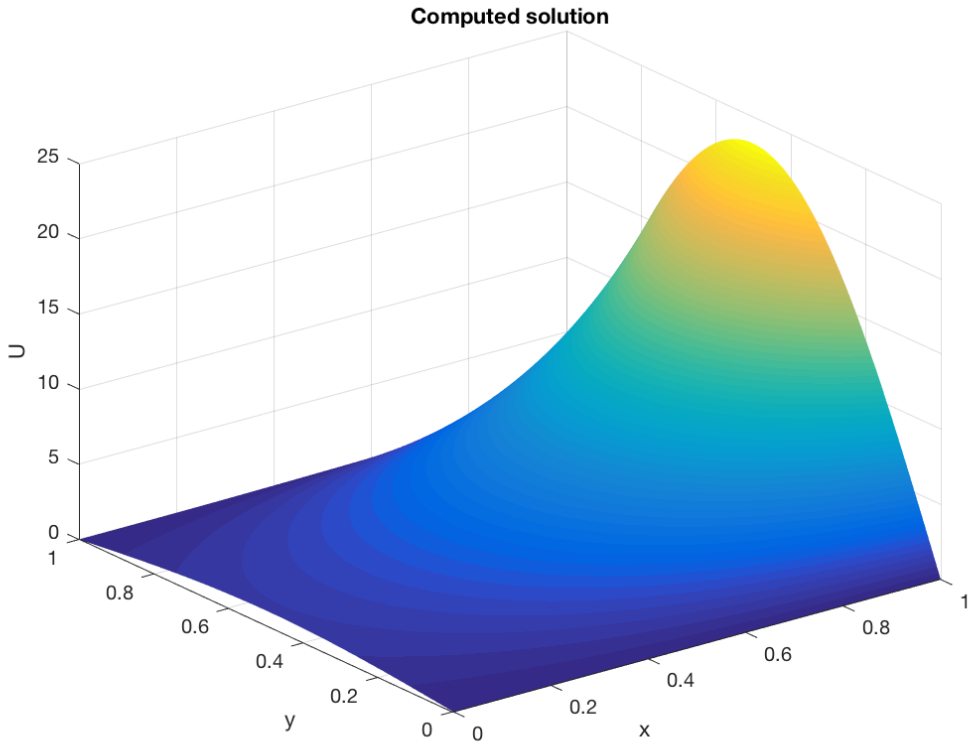

Figure 15: Result of 23 Vcycles with the default parameters

A Appendix

A.1 2-point Boundary Value Problems

```
function U = NewtonsMethod(FunJac,Uinit,h,tol,maxit)
U = Uinit;
[G,J] = FunJac(U,h);
k = 0;
while ((k < maxit) && (norm(G,'inf') > tol))
    k = k+1;
    U(2:end-1) = U(2:end-1) - J\G;
    [G,J] = FunJac(U,h);
end
```

```
function [G,J] = FunJac(U,h)
% This function builds the vector G and its Jacobian to solve the BVP
    using
% Newton's method.

G = 0.01/h^2*( U(1:end-2) - 2*U(2:end-1) + U(3:end)) ...
    + U(2:end-1).*((U(3:end) - U(1:end-2))/(2*h) - 1);

J = 1/h^2 * (diag(-2*0.01 + 0.5*h*(U(3:end) - U(1:end-2))-h^2) ...
    + diag(0.01 - 0.5*h*U(3:end-1),-1) + diag(0.01 + 0.5*h*U(2:end-2)
        ,1));

end
```

```
function [c,info] = bisection(odefun,ts,epsilon,a,b,tol,maxit)
rc = tol+1;
k = 1;

Ua0 = [-1; a];
[ta,Ua] = ode45(odefun,ts,Ua0,[],epsilon);
ra = Ua(end,1)-1.5;

while ((k < maxit) && (abs(rc(end)) > tol))
    c = (a+b)/2;
    Uc0 = [-1; c];
    [tc,Uc] = ode45(odefun,ts,Uc0,[],epsilon);
    rc(k) = Uc(end,1)-1.5;
    if (sign(rc(end)) == sign(ra))
        a = c;
    else
        b = c;
    end
    k = k+1;
end
```

```
info.r = abs(rc);
info.iter = k-1;
end
```

```
function [sigma,info] = NewtonShooting(odefun,ts,epsilon,sigma0,tol,
    maxit)

r = tol+1;
k = 1;
sigma = sigma0;

while ((k < maxit) && (abs(r(end)) > tol))
    w0 = [-1; sigma; 0; 1];
    [t,W] = ode45(odefun,ts,w0,[],epsilon);
    r(k) = W(end,1)-1.5;
    sigma = sigma - r(end)/W(end,3);
    k = k+1;
end

info.r = abs(r);
info.iter = k-1;
end
```

```
function [sigma,info] = secant(odefun,ts,epsilon,sigma0,tol,maxit)

k = 1;
r = tol+1;
mu = 1e-4;
sigma = sigma0;

while ((k < maxit) && (abs(r(end)) > tol))
    U0 = [-1; sigma];
    [t1,U1] = ode45(odefun,ts,U0,[],epsilon);
    U0e = [-1; sigma+mu];
    [t2,U2] = ode45(odefun,ts,U0e,[],epsilon);
    duds = (U2(end,1) - U1(end,1))/mu;
    r(k) = U1(end,1)-1.5;
    sigma = sigma - r(end)/duds;
    k = k+1;
end
info.r = abs(r);
info.iter = k-1;
end
```

```
close all
%% 2-POINT BVPs

%% Solve using bvp4c
odefun = @(t,x,epsilon) [x(2); -x(1)*(x(2)-1)/epsilon];
```

```
bcfun = @(xa,xb,epsilon) [xa(1)+1; xb(1)-1.5];

options = bvpset('reltol',1e-2,'abstol',1e-2);
options2 = bvpset('reltol',1e-10,'abstol',1e-10);

solinit = bvpinit(linspace(0,1,10),[0 0]);

epsilon = 0.01;
sol = bvp4c(odefun,bcfun,solinit,options,epsilon);
sol2 = bvp4c(odefun,bcfun,solinit,options2,epsilon);
plot(sol2.x,sol2.y(1,:),'linewidth',1.6)
%% Solve using Newton's method
n = length(sol.x);
n = 100;
t = linspace(0,1,n)';
% Aproximate solution 2.105
omega = 0.5*(0-1+1.5+1);
tm = 0.5*(0+1+1-1.5);
Uinit = t-tm+omega*tanh(omega*(t-tm)/(2*0.01));
%plot(t,Uinit,'--','linewidth',1.6)
hold on


h = 1/(n+1);
tol = 1e-5;
maxit = 1000;
U = NewtonsMethod(@FunJac,Uinit,h,tol,maxit);
plot(t,U,'linewidth',1.6)

xlabel('t')
ylabel('U')
legend('bvp4c','Newton','location','southeast')
print('sol2','-dpng')
%% Solve using the single shooting method

%% Bisection method
ts = [0 1];
epsilon = 0.01;

a = 1;
Ua0 = [-1; a];
[ta,Ua] = ode45(odefun,ts,Ua0,[],epsilon);

b = 5;
Ub0 = [-1; b];
[tb,Ub] = ode45(odefun,ts,Ub0,[],epsilon);
t1 = cputime;
for i = 1:1
[c,info1] = bisection(odefun,ts,epsilon,a,b,tol,maxit);
```

```
end
tbisection = (cputime-t1)/100;

Uc0 = [-1; c];
[tc,Uc] = ode45(odefun,ts,Uc0,[],epsilon);

figure
plot(ta,Ua(:,1),tb,Ub(:,1),'linewidth',1.6)
hold on
plot(tc,Uc(:,1),'linewidth',1.6)

xlabel('t')
ylabel('U')
legend('U_a','U_b','U_c','location','northwest')
print('bisection','-dpng')
%% Sensitivity analysis finite difference approximation
% Forward
epsilon = 0.001;
mu = 1e-7;
U0e = [-1; c+mu];
[te,Ue] = ode45(odefun,ts,U0e,[],epsilon);
duds = (Ue(end,1) - Uc(end,1))/mu;
% Backward
mu = 1e-7;
U0e = [-1; c-mu];
[te,Ue] = ode45(odefun,ts,U0e,[],epsilon);
duds2 = (Uc(end,1) - Ue(end,1))/mu;

% Sensitivity analysis analytical solution
odefun2 = @(t,w,epsilon) [w(2); -w(1)*(w(2)-1)/epsilon; w(4); (-w(2)
    +1)/epsilon*w(3)-w(1)/epsilon*w(4)];
w0 = [-1; c; 0; 1];
[t,W] = ode45(odefun2,ts,w0,[],epsilon);


%% Plot solution as a function of sigma
sigma = linspace(0.9999,1.01,1000);
Uend = zeros(1000,1);
for i=1:1000
    U0 = [-1; sigma(i)];
    [t,U] = ode45(odefun,ts,U0,[],epsilon);
    Uend(i) = U(end,1);
end
r = Uend - 1.5;
subplot(1,2,1)
plot(sigma,r)
xlabel('\sigma')
ylabel('r(\sigma)')
subplot(1,2,2)
```

```
plot(sigma,r)
xlabel('\sigma')
ylabel('r(\sigma)')
axis([1 1.00001 -2 0.2]);
print('usigma','-dpng')
%% Newton's method
sigma0 =3;
tol = 1e-2;
maxit = 100;
epsilon = 0.01;
odefun2 = @(t,w,epsilon) [w(2); -w(1)*(w(2)-1)/epsilon; w(4); (-w(2)
    +1)/epsilon*w(3)-w(1)/epsilon*w(4)];

t1 = cputime;
for i = 1:1000
[sigma,info2] = NewtonShooting(odefun2,ts,epsilon,sigma0,tol,maxit);
end
tnewton = (cputime-t1)/1000;

plot(1:info2.iter,info2.r,'linewidth',1.6)
hold on
plot(1:info1.iter,info1.r,'linewidth',1.6)
axis([1 7 0 0.6])
xlabel('iteration')
ylabel('residual')
legend('Newton','bisection')
print('rate','-dpng')
%% Finite difference approximation/Secant Method
sigma0 = 3;
t1 = cputime;
for i=1:1000
[sigma,info3] = secant(odefun,ts,epsilon,sigma0,tol,maxit);
end
tsecant = (cputime-t1)/1000;
```

A.2 9-point Laplacian

```
function A = poisson5(m)

e = ones(m,1);
S = spdiags([e -2*e e], [-1 0 1], m, m);
I = speye(m);
A = kron(I,S)+kron(S,I);
A = (m+1)^2*A;

end
```

```
function A = poisson9(m)

```

```
% TODO: This approach has problem with overlapping corners, but it
    can be used
% If there is time try this one also, just modify the S matrix to
    account
% for the corners in the first and last row
% e = ones(m,1);
% S = spdiags([e 16*e -30*e 16*e e], [-2 -1 0 1 2], m, m);
% I = speye(m);
% A = kron(I,S)+kron(S,I);

% classical 9-point stencil (square)
h = 1/(m+1);
e = ones(m,1);
S = -spdiags([e 10*e e], [-1 0 1], m, m);
I = spdiags([-1/2*e e -1/2*e], [-1 0 1], m, m);
A = kron(I,S)+kron(S,I);
A = A/(6*h^2);
end
```

```
function F = form_rhs5(m,f,u)

h = 1/(m+1);

% form RHS for 5-point Laplacean

[X,Y] = meshgrid(linspace(0,1,m+2), linspace(0,1,m+2));
U = u(X,Y);

Iint = 2:m+1;
Jint = 2:m+1;
Xint = X(Iint,Jint);
Yint = Y(Iint,Jint);

F = f(Xint,Yint);

F(:,1) = F(:,1) - U(2:m+1,1)*(1/h^2);
F(:,m) = F(:,m) - U(2:m+1,m+2)*(1/h^2);
F(1,:) = F(1,:) - U(1,2:m+1)*(1/h^2);
F(m,:) = F(m,:) - U(m+2,2:m+1)*(1/h^2);
F = reshape(F,m*m,1);

end
```

```
function [ F ] = form_rhs( m, f, u )
% form RHS for 9-point Laplacean
h = 1/(m+1);
hmult = 1/(6*h^2);
%hmult = 1; % used for debug with symbolic vars

```

```
[X,Y] = meshgrid(linspace(0,1,m+2), linspace(0,1,m+2)); % account for
    outside of the grid
Iint = 2:m+1;
Jint = 2:m+1;
Xint = X(Iint,Jint);
Yint = Y(Iint,Jint);

F = zeros(m);
% boundaries
U = u(X,Y);
U(2:m+1,2:m+1) = 0;
% Uncomment to test with symbolic variables to see if Matlab code
    matches the calculations on paper
% U = sym('U%d%d', [m+2 m+2]);
% U(2:m+1,2:m+1) = 0;
% F = sym('F%d%d', [m m]);

% common for all (similar to 5-point L RHS)
%   - x -
%   x - x
%   - x -
F(:,1) = F(:,1) - hmult * 4 * U(2:m+1, 1    ); % left column
F(:,m) = F(:,m) - hmult * 4 * U(2:m+1, m+2  ); % right column
F(1,:) = F(1,:) - hmult * 4 * U(1,     2:m+1); % top row
F(m,:) = F(m,:) - hmult * 4 * U(m+2,   2:m+1); % bottom row
% account for corner points of the 9-point L stencil
%   x - x
%   - - -
%   x - x
for k=1:m
    F(k,1) = F(k,1) - hmult * U(k,1) - hmult * U(k+2,1);
    F(k,m) = F(k,m) - hmult * U(k,m+2) - hmult * U(k+2,m+2);

    F(1,k) = F(1,k) - hmult * U(1,k) - hmult * U(1,k+2);
    F(m,k) = F(m,k) - hmult * U(m+2,k) - hmult * U(m+2,k+2);
end
% account for the contribution from corners of the grid that was
    added
% twice in the previous step
F(1,1) = F(1,1) + hmult * 1 * U(1,1);
F(m,m) = F(m,m) + hmult * 1 * U(m+2,m+2);
F(m,1) = F(m,1) + hmult * 1 * U(m+2,1);
F(1,m) = F(1,m) + hmult * 1 * U(1,m+2);
% finalize
FC = reshape( poisson5(m+2)*reshape(f(X,Y),(m+2)^2,1) , m+2, m+2);
F = f(Xint,Yint) + F + ((1/12)*h^2)*FC(Iint,Jint);

F = reshape(F, m*m, 1);

```

```
53 end
```

```
1  close all;
2  %% Setup
3  a = 0;
4  b = 1;
5  m = 30;
6  h = (b-a)/(m+1);
7  x = linspace(a,b,m+2);   % grid points x including boundaries
8  y = linspace(a,b,m+2);   % grid points y including boundaries
9
10
11 [X,Y] = meshgrid(x,y);      % 2d arrays of x,y values
12
13 Iint = 2:m+1;              % indices of interior points in x
14 Jint = 2:m+1;              % indices of interior points in y
15 Xint = X(Iint,Jint);       % interior points
16 Yint = Y(Iint,Jint);
17
18 fdummy = @(x,y) 0*x.*y;
19
20 %% Dummy test
21 utrue = @(x,y) exp(x+y/2);
22 f = @(x,y) 1.25 * exp(x+y/2);
23 Utrue = utrue(X,Y);
24
25 RHS = form_rhs(m,f,utrue);
26 A = poisson9(m);
27 U = Utrue;
28 U(Iint,Jint) = reshape(A\RHS(:), m, m);
29
30 figure(1);
31 surf(X,Y,Utrue);
32
33 figure(2);
34 surf(X, Y, U);
35
36 %% Test case 0
37 u0 = @(x,y) sin(4*pi*(x+y))+cos(4*pi*x.*y);
38 U0 = u0(X,Y);
39 % given by matlab simplify(laplacian() or diff(...,2)+diff(...,2))
40 f0 = @(X,Y) -16 * (pi ^ 2) * (...
41     cos((4 * pi * X .* Y)) .* (X .^ 2)...
42     + cos((4 * pi * X .* Y)) .* (Y .^ 2)...
43     + 2 * sin((4 * pi * (X + Y)))...
44 );
45
46 RHS0 = form_rhs(m, f0, u0);
47 A = poisson9(m);
```

```
%U0calc = U0;
U0calc = reshape( A\RHS0(:), m, m );

figure(1)
surf(Xint,Yint,U0(Iint,Jint));

figure(2)
surf(Xint,Yint,U0calc);

%% Test case 0 - global error

Gerr = [];
Grange = [15 100 500 1000 30];
for k=Grange
    m = k;
    h = 1/(m+1);
    x = linspace(0,1,m+2);
    y = linspace(0,1,m+2);
    [X,Y] = meshgrid(x,y);
    Iint = 2:m+1;
    Jint = 2:m+1;
    Xint = X(Iint,Jint);
    Yint = Y(Iint,Jint);

    U0 = u0(X,Y);
    RHS0 = form_rhs(m, f0, u0);
    A = poisson9(m);
    U0calc = reshape( A\RHS0(:), m, m );

    Gerr = [Gerr(:); max(max( abs( U0(Iint,Jint)-U0calc ) ))];
    fprintf('Finished calculating for m=%d\n', m)
end
figure(3)
hs = 1./((Grange+ones(size(Grange)))); % h = 1/(m+1)
loglog(hs, Gerr, hs, hs.^(4), hs, hs.^(2), 'LineWidth', 2.4)
legend({'Global error', '$\mathcal{O}(h^4)$', '$\mathcal{O}(h^2)$'},
    'Location', 'SouthEast', 'FontSize', 24, 'Interpreter', 'latex')
fprintf('Test case 0 done!\n')

%% Test case 1 (run before global errors for faster comp)
u1 = @(x,y) x.^2 + y.^2;
U1 = u1(Xint,Yint);
f1 = @(x,y) 4*x.^0.*y.^0;

RHS1 = form_rhs(m, f1, u1); % f(x,y) = 4
U1calc = reshape( poisson9(m)\RHS1(:), m, m );

figure(1)
surf(Xint,Yint,U1);
```

```
96
97  figure(2)
98  surf(Xint,Yint,U1calc);
99
100 %% Test case 1 - global error
101
102 Gerr = [];
103 Grange = 10:10:350;
104 for k=Grange
105     m = k;
106     h = 1/(m+1);
107     x = linspace(0,1,m+2);
108     y = linspace(0,1,m+2);
109     [X,Y] = meshgrid(x,y);
110     Iint = 2:m+1;
111     Jint = 2:m+1;
112     Xint = X(Iint,Jint);
113     Yint = Y(Iint,Jint);
114
115     U = u1(X,Y);
116     RHS = form_rhs(m, f1, u1);
117     A = poisson9(m);
118     Ucalc = reshape( A\RHS(:), m, m );
119
120     Gerr = [Gerr(:); max(max( abs( U(Iint,Jint)-Ucalc ) ))];
121     fprintf('Finished calculating for m=%d\n', m)
122 end
123 figure(3)
124 hs = 1./((Grange+ones(size(Grange)))); % h = 1/(m+1)
125 loglog(hs, Gerr, '-x', hs, hs.^(4), hs, hs.^(2), 'LineWidth', 2.4)
126 legend({'Global error', '$\mathcal{O}(h^4)$', '$\mathcal{O}(h^2)$'},
        'Location', 'SouthEast', 'FontSize', 24, 'Interpreter', 'latex')
127 fprintf('Test case 1 done!\n')
128
129 %% Test case 2 (run before global errors for faster comp)
130 u2 = @(x,y) sin(2*pi*abs(x - y).^(2.5));
131 U2 = u2(X,Y);
132 f2 = @(x1,y1) 5*pi*abs(x1 - y1).^(1/2).*sign(x1 - y1).^2.*(...
133     3*cos(2*pi*abs(x1 - y1).^(5/2))...
134     - 10*pi*abs(x1 - y1).^(5/2).*sin(2*pi*abs(x1 - y1).^(5/2))...
135 );
136 U2calc = U2;
137 U2calc(Iint,Jint) = reshape( poisson9(m)\form_rhs(m, f2, u2), m, m );
138
139 figure(1)
140 surf(X,Y,U2);
141
142 figure(2)
143 surf(X,Y,U2calc);
```

```
144
145 %% Test case 2 - global error
146
147 Gerr = [];
148 Grange = [15 30 100 500 1000];
149 for k=Grange
150     m = k;
151     h = 1/(m+1);
152     x = linspace(0,1,m+2);
153     y = linspace(0,1,m+2);
154     [X,Y] = meshgrid(x,y);
155     Iint = 2:m+1;
156     Jint = 2:m+1;
157     Xint = X(Iint,Jint);
158     Yint = Y(Iint,Jint);
159
160     U = u2(X,Y);
161     RHS = form_rhs(m, f2, u2);
162     A = poisson9(m);
163     Ucalc = reshape( A\RHS(:), m, m );
164
165     Gerr = [Gerr(:); max(max( abs( U(Iint,Jint)-Ucalc ) ))];
166     fprintf('Finished calculating for m=%d\n', m)
167 end
168 figure(3)
169 hs = 1./((Grange+ones(size(Grange)))); % h = 1/(m+1)
170 loglog(hs, Gerr, hs, hs.^(4), hs, hs.^(2), 'LineWidth', 2.4)
171 legend({'Global error', '$\mathcal{O}(h^4)$', '$\mathcal{O}(h^{2})$'
        }, 'Location', 'SouthEast', 'FontSize', 24, 'Interpreter', 'latex'
        )
172 fprintf('Test case 2 done!\n')
```

A.3 Iterative solvers in 2d

```
1  function [ AU ] = Amult( U, m )
2  %AMULT for Vcycle
3
4  h = 1/(m+1);
5
6  % convert to a squared matrix for convenience
7  U=reshape(U,m+2,m+2);
8  Iint = 2:m+1;
9  Jint = 2:m+1;
10
11 AU = U;
12 AU(Iint,Jint) = (1/h^2)*( U(Iint-1,Jint) + U(Iint+1,Jint) + U(Iint,
       Jint-1) + U(Iint, Jint+1) - 4*U(Iint,Jint) );
13
14 % reshape into a column vector -Au
```

```
AU = -AU(:);

end
```

```
function Unew=smooth(U,omega,m,F)
% return m+2 by m+2
    h = 1/(m+1);
    U = reshape(U,m+2,m+2);
    F = reshape(F,m+2,m+2);
    Iint = 2:m+1;
    Jint = 2:m+1;
    U(Iint,Jint) = (1-omega)*U(Iint,Jint) + omega/4*( U(Iint-1,Jint)
        + U(Iint+1,Jint) + U(Iint,Jint-1) + U(Iint, Jint+1) - h^2*F(
        Iint,Jint));
    Unew = U(:);
end
```

```
function [ Rc ] = coarsen( R, m )
% Restriction
% see:
% https://cn.inside.dtu.dk/cnnet/filesharing/download/10824fda-2212-4
    f3c-8baf-54cc461f365f
% slide 9

mc = (m-1)/2;
R = reshape(R,m+2,m+2);
% idx = 2:2:length(R);
% Rc = 1/4*(R(idx+1,idx-1) + R(idx+1,idx+1) + R(idx-1, idx-1) + R(idx
    -1, idx+1));

Rc = R(1:2:end,1:2:end);

% R = reshape(R,m,m);
% ind = 2:2:length(R);
%
% Rc = ( 4*R(ind,ind) + ...
%                     2*(R(ind-1,ind)+R(ind+1,ind)+ ...
%                        R(ind,ind-1)+R(ind,ind+1)) + ...
%                       (R(ind-1,ind-1)+R(ind-1,ind+1)+ ...
%                        R(ind+1,ind-1)+R(ind+1,ind+1)) )/16;

end
```

```
function [ R ] = interpolate( Rc, m )

mc = (m-1)/2;
Rc = reshape(Rc, mc+2, mc+2);
R = zeros(m+2,m+2);

```

```
% coarsed pts
R(1:2:end, 1:2:end) = Rc;
% inner points
R(2:2:m+1, 2:2:m+1) = 1/4*(Rc(1:end-1, 1:end-1) ...%topleft
    + Rc(2:end,2:end)...%bottomright
    + Rc(1:end-1,2:end)...%top right
    + Rc(2:end,1:end-1)...% bottom left
    );
% add above and below
R(2:2:end-1,1:2:end) = 1/2*(Rc(1:end-1,:) +  Rc(2:end,:));
% left and right
R(1:2:end,2:2:end-1) = 1/2*(Rc(:,1:end-1) +  Rc(:,2:end));

% for k=1:mc
%     for l=1:mc
%         % center
%         R(2*k, 2*l) = R(2*k, 2*l) + Rc(k,l);
%         % top left
%         R(2*k-1, 2*l-1) = R(2*k-1, 2*l-1) + Rc(k,l)/4;
%         % top right
%         R(2*k-1, 2*l+1) = R(2*k-1, 2*l+1) + Rc(k,l)/4;
%         % bottom left
%         R(2*k+1, 2*l-1) = R(2*k+1, 2*l-1) + Rc(k,l)/4;
%         % bottom right
%         R(2*k+1, 2*l+1) = R(2*k+1, 2*l+1) + Rc(k,l)/4;
%         % left
%         R(2*k, 2*l-1) = R(2*k, 2*l-1) + Rc(k,l)/2;
%         % top
%         R(2*k-1, 2*l) = R(2*k-1, 2*l) + Rc(k,l)/2;
%         % right
%         R(2*k, 2*l+1) = R(2*k, 2*l+1) + Rc(k,l)/2;
%         % bottom
%         R(2*k+1, 2*l) = R(2*k+1, 2*l) + Rc(k,l)/2;
%     end
% end

end
```

```
function [ F ] = form_rhsV( m, f, u )
%FORM_RHSV Summary of this function goes here
%   Detailed explanation goes here

h = 1/(m+1);
[X,Y] = meshgrid(linspace(0,1,m+2), linspace(0,1,m+2));
Iint = 2:m+1;
Jint = 2:m+1;
Xint = X(Iint,Jint);
Yint = Y(Iint,Jint);

```

```
F = u(X,Y);
F(Iint,Jint) = f(Xint,Yint);

F=F(:);

end
```

```
function Unew=Vcycle(Ufull,omega,nsmooth,m,F)
% Approximately solve: A*U = F
h=1.0/(m+1);
l2m=log2(m+1);
assert(l2m==round(l2m));

assert(length(Ufull)==(m+2)^2);

if(m==1)
    U = reshape(Ufull, 3, 3);
    Uval = 1/4 * (U(1,2) + U(2,1) + U(2,3) + U(3,2) - h^2*F(5));
    Unew = U;
    Unew(2,2) = Uval;
    Unew = Unew(:);
else
    %fprintf('Running the recursive part m: %d \n', m);

    Unew = Ufull;
    for numSmooth=1:nsmooth
        Unew = smooth(Unew,omega,m,F(:));
    end

    R = F(:) + Amult(Unew(:),m); % Amult returns -Au, so +

    Rcoarse = coarsen(R,m);

    mc=(m-1)/2;
    Ecoarse=Vcycle(zeros((mc+2)^2,1),omega,nsmooth,mc,-Rcoarse);

    %fprintf('Returning from the recursive part m: %d \n', m);

    E = interpolate(Ecoarse(:),m);
    E = reshape(E, m+2,m+2);
    Err = zeros(size(E));
    Err(2:end-1,2:end-1) = E(2:end-1,2:end-1);

    Unew = Unew(:) + Err(:);

    for numSmooth=1:nsmooth
        Unew = smooth(Unew,omega,m,F);
    end

```

```
    Unew = Unew(:);
end

end
```

```
function plotU(m,U)
h=1/(m+1);
x=linspace(0,1,m+2);
y=linspace(0,1,m+2);
[X,Y]=meshgrid(x,y);
surf(X, Y, reshape(U,[m+2,m+2]));
shading interp;
title('Computed solution');
xlabel('x');
ylabel('y');
zlabel('U');
end
```

```
function [ AU ] = mAmult( U, m )
%AMULT Summary of this function goes here
%   Detailed explanation goes here

h = 1/(m+1);

% convert to a squared matrix for convenience
U=reshape(U,m,m);

AU=4*U;
AU(1:m-1,:) = AU(1:m-1,:) - U(2:m,:);
AU(2:m,:) = AU(2:m,:) - U(1:m-1,:);
AU(:,2:m) = AU(:,2:m) - U(:,1:m-1);
AU(:,1:m-1) = AU(:,1:m-1) - U(:,2:m);

% reshape into a column vector -Au
AU = AU(:)/h^2;

end
```

```
m = 30;
h = 1/(m+1);
[X,Y] = meshgrid(0:h:1,0:h:1);
Xint = X(2:m+1,2:m+1);
Yint = Y(2:m+1,2:m+1);

% Change to -Au=-F (`pcg` requires positive definite system)
f0 = @(X,Y) -16 * (pi ^ 2) * (...
    cos((4 * pi * X .* Y)) .* (X .^ 2)...
    + cos((4 * pi * X .* Y)) .* (Y .^ 2)...
    + 2 * sin((4 * pi * (X + Y)))...
```

```
);
b = -f0(Xint,Yint);
Afun = @(u) mAmult(u,m); % already returned as -Au

u = pcg(Afun,b(:),h^2,100);
% Check error
max(max(abs(Amult(u,m) - -1*poisson5(m)*u)))

figure(1)
surf(Xint,Yint,reshape(u,m,m));
```

```
%% 3.2 Under/Over-relaxed Jacobi Smoothing

% eigenvalues expression 4.90 REVIEW
feig = @(p,q,h,omega) (1-omega)+omega.*(cos(p.*pi*h)+cos(q.*pi*h));

m = 1000;
h = 1/(m+1);

% obtain all combinations of p and q
p = m/2:m;
q = m/2:m;
[P,Q] = meshgrid(p,q);

% memory allocation
maxeig = zeros(100,1);

% for each value of omega find the maximum eigenvalue
omega = linspace(0,2,1000);
for i = 1:1000
    eig = feig(P,Q,h,omega(i));
    maxeig(i) = max(max(abs(eig)));
end
% choose omega that makes maxeig minimum
[minmax,idx] = min(maxeig);
omegaopt = omega(idx);
plot(omega,maxeig,'linewidth',1.6)
hold on
plot(omegaopt,minmax,'ro','linewidth',1.6)
legend('maximum eigenvalue','optimal \omega','location','northwest')
xlabel('\omega')
ylabel('max(\rho_{p,q}(\omega))','interpreter','tex')
print('omegaopt','-dpng')
```

```
% exact solution and RHS
u=@(x,y) exp(pi*x).*sin(pi*y)+0.5*(x.*y).^2;
f=@(x,y) x.^2+y.^2;

m=2^7-1;
```

```
6
7  [X,Y] = meshgrid(linspace(0,1,m+2), linspace(0,1,m+2));
8
9  Uall = u(X,Y);
10 Uall(2:end-1, 2:end-1) = 0;
11
12 U = Uall(:);
13
14 F = form_rhsV(m,f,u);
15
16 epsilon = 1.0E-10;
17 omega = 4/5;
18
19 for i=1:100
20     R = F(:) + Amult(U,m); % Au = F -> R = F - Au
21     fprintf('*** Outer iteration: %3d, rel. resid.: %e\n', ...
22         i, norm(R,2)/norm(F,2));
23     if(norm(R,2)/norm(F,2) < epsilon)
24         break;
25     end
26     U=Vcycle(U,omega,10000,m,F);
27     plotU(m,U);
28 end
```