Math 567 Final Project

Gurpal Singh

December 13, 2017

Abstract

In this project, three different methods will be implemented to approximate the solution to a one-dimensional heat-diffusion problem in a rod. First, the heat-diffusion solution will be solved using the central difference discretization in space and Forward-Euler method to timestep. Next, an iterative/direct method will be used. The central difference discretization will still be used for space, but a backward Euler scheme will be implemented to time-step the solution. Finally, the last approach to approximate the solution involves an implementation of the Runge-Kutta Chebyshev functions. All three methods are compared to the true solution to verify the order of accuracy of each method. Lastly, the pros and cons of each method are explored.

1 Introduction

The one-dimensional heat diffusion problem is an ideal case to explore the accuracy of different numerical methods. The exact solution is known and we can compare our numerical schemes. Specifically, the forward euler, backward euler, and Runge-Kutta Chebychev methods will be used. We will see the benefits and consequences of using explicit, direct, and muliti-stage methods. In the following sections, each method will be discussed in detail.

2 The Physical Problem

The one-dimensional heat diffusion problem can be written as a partial differential equation as follows.

$$u_t = \kappa u_{xx} \tag{1}$$

2.1 Problem Set-Up

The 1-D rod used for our test case is shown in figure 1. There are Dirichlet Boundary Conditions applied at each end. The left and right ends of the rod are each imposed with the condition $T = 0$. The thermal conductivity and the length of the rod are both assumed to be equal to 1. The rod is imposed with the initial condition, $u(x, 0)$ where u is the exact solution from section 2.2.

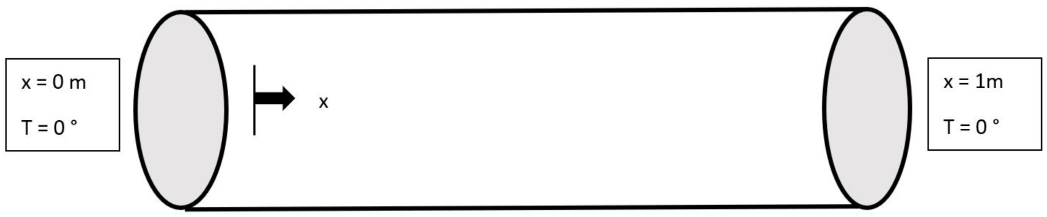


Figure 1: 1-D Rod with Dirichlet Boundary Conditions.

2.2 Exact Solution

To test the accuracy of the methods, we must pick an exact solution such that it satisfies equation 1 above. It happens that the following exact solution fits our requirements.

$$u(x,t) = e^{-m^2\pi^2 t} sin(m\pi x) \tag{2}$$

Differentiating with respect to time to get u_t

$$u_t = -m^2\pi^2 e^{-m^2\pi^2 t} sin(m\pi x) \tag{3}$$

Differentiating with respect to x to get u_x

$$u_x = m\pi * e^{-m^2\pi^2 t} cos(m\pi x) \tag{4}$$

Differentiating with respect to x again to get u_{xx}

$$u_{xx} = -m^2\pi^2 e^{-m^2\pi^2 t} sin(m\pi x) \tag{5}$$

Comparing equations 3 and 5, we can see that u_t does indeed equal u_{xx}.

3 Spatial Discretization

The spatial discretization was done via central difference for all three methods. An implantation of the spatial discretization for the one-dimensional heat diffusion problem is given below.

$$u_{xx} = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} \tag{6}$$

In the implementation it is more efficient to create a matrix "A" with a spatial discretization for grid point, which results in system of equations. Since the boundary conditions never change over time in this case, only the interior points are discretized. This means that a one-dimensional domain with n points would have an "A" matrix of size $(n-2)$ x $(n-2)$

The linear system one has to solve is:

$$AU = F \tag{7}$$

where "U" is the vector of temperature solutions at each grid point and F is the right hand side.

The "A" matrix for the interior points (boundary points $x = 0$ and $x = L$ not included) can be constructed as follows:

$$A = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}$$

4 Method 1: Explicit Method Forward Euler

4.1 Implementation

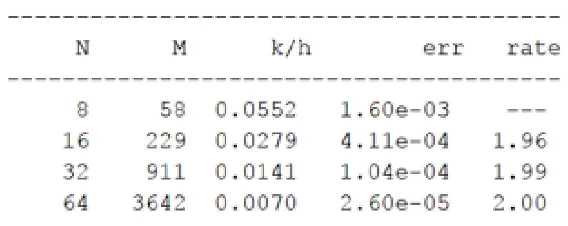
The Forward Euler method is an explicit method meaning it does not involve a function evaluation at the time step we are trying to solve for. Forward Euler is a first order method meaning its error is $\mathcal{O}(h)$. The Forward Euler scheme can be written as follows.

$$U^{n+1} = U^n + kf(U^n) \tag{8}$$

Since f is the right-hand had side and we know that $\frac{1}{h^2}AU^n = F^n$, where F is the right-hand side evaluated at each grid point. We can then write the Forward Euler scheme in matrix form.

$$U^{n+1} = U^n + \frac{k}{h^2}AU^n \tag{9}$$

This is how the scheme is implemented in the Matlab.



```
----------------------------------------
     N       M       k/h        err   rate
----------------------------------------
     8      58  0.0552  1.60e-03    ---
    16     229  0.0279  4.11e-04   1.96
    32     911  0.0141  1.04e-04   1.99
    64    3642  0.0070  2.60e-05   2.00
```

Figure 2: Forward Euler Matlab Convergence Study Results.

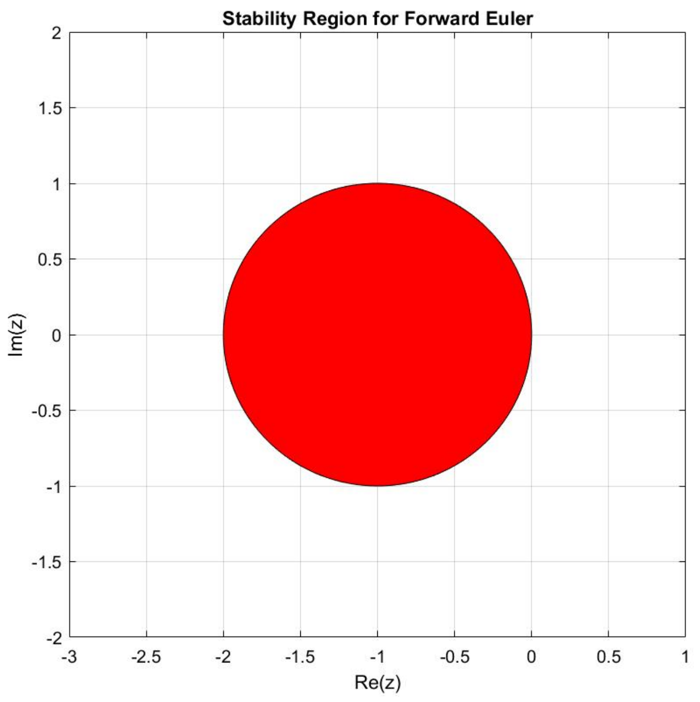


Figure 3: The stability region for the Forward Euler method is the disk centered at (0,-1) and radius 1.

4.2 Results

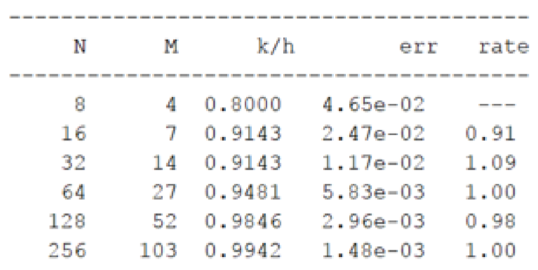
The results from the Forward Euler in time and central difference in space from the convergence study conducted in Matlab is shown in Figure 2. For this method k, the time step size is set as $k = 0.45h^2$. It is clear from the table of results that as we increase the number of intervals between the gridpoints, N, the number of timesteps required to converge to a solution increases dramatically. The results also show that the Forward-Euler method produces a second order accurate approximation.

4.3 Pros and Cons

As you saw above, we set $k = 0.45h^2$. This is a direct result of the stability requirements needed for the Forward Euler method. To show the stability region, we will use the stability polynomial. Finding the roots of the stability polynomial for linear multistep methods, allows us to visually graph the region of stability. For Forward Euler we have:

$$\pi(\zeta; z) = \zeta - (1 + \zeta) \tag{10}$$

Forward Euler has the severe restriction that $\frac{k}{h^2} \leq \frac{1}{2}$, where k is the time step size and h is the grid spacing size between each point. Thus, Forward Euler requires fairly small time steps for it to be a stable method.



```
--------------------------------------------
      N      M      k/h        err    rate
--------------------------------------------
      8      4   0.8000   4.65e-02     ---
     16      7   0.9143   2.47e-02    0.91
     32     14   0.9143   1.17e-02    1.09
     64     27   0.9481   5.83e-03    1.00
    128     52   0.9846   2.96e-03    0.98
    256    103   0.9942   1.48e-03    1.00
```

Figure 4: Backward Euler Matlab Convergence Study Results.

5 Method 2: Direct Method Backward Euler

5.1 Implementation

The Backward Euler method is an implicit method since it requires an evaluation of the function at the updated time step, $f(U^{n+1})$. It would require the additional operation of finding the zero at each iteration, which can be approximated by iterative methods such as Newton's method or a direct solve in linear algebra. This is also a first-order accurate method.

$$U^{n+1} = U^n + kf(U^{n+1}) \tag{11}$$

In the Matlab implementation, the backward-Euler method is vectorized and a direct solve is used to solve for the updated time step. Recall, we can write $f(U^{n+1}$ as AU^{n+1}.

$$U^{n+1} = U^n + \frac{k}{h^2}AU^{n+1} \tag{12}$$

$$U^{n+1} - \frac{k}{h^2}AU^{n+1} = U^n \tag{13}$$

$$U^{n+1}(I - \frac{k}{h^2}A) = U^n \tag{14}$$

$$U^{n+1} = U^n/(I - \frac{k}{h^2}A) \tag{15}$$

5.2 Results

The results from the Backward-Euler scheme in time and central difference discretization in space from the convergence study conducted in Matlab is shown in Figure 4. For this method k, the time step size is set as $k = h$. Thus, the number of time steps required, M, is much less than the Forward-Euler method. However, the downside to this method is that it is only first order accurate.

5.3 Pros and Cons

Notice, that we are able to take a lot larger time step, in the backward-Euler method. However, this comes at cost. With the backward-Euler method, we now have to do a direct solve at each time step. This increases the computational cost but reduces the total number of time steps we need to take compared to the forward-Euler method.

The stability function for the backward-Euler method can be written as follows.

$$\pi(\zeta; z) = (1 - z)\zeta - 1 \tag{16}$$

Finding the roots, we get, $|1 - z| \geq 1$. This indicates that the stability region for the backward-Euler method is the exterior of the disk of radius 1 centered at z = 1. It is interesting to note that the backward-Euler's stability region is the opposite of the forward-Euler's stability region.

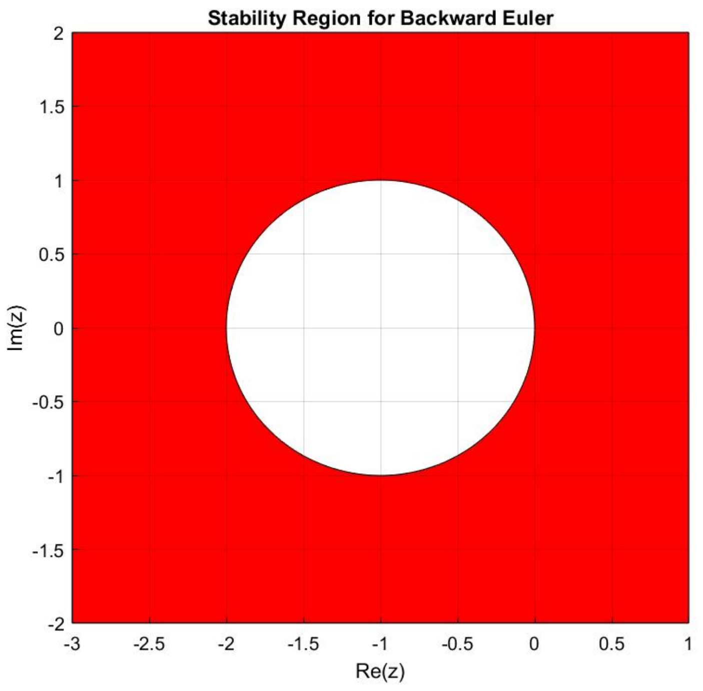


Figure 5: The stability region for the Backward Euler method is everything outside of the disk centered at (0,-1) and radius 1.

6 Method 3: Runge-Kutta + Chebyschev Functions

6.1 Implementation

The Runge-Kutta-Chebyshev methods are useful for problems where the eigenvalues of the Jacobian matrix are located on the negative real axis. This is often common when the grid resolution becomes finer. Runge-Kutta-Chebyshev methods are unique in the sense that, the region of stability can be extended along the negative real axis by adding more stages to the method. We will be implementing a second order Runge-Kutta-Chebyshev method. The function to compute the parameters for the RKC method was provided by Dr. Calhoun. The RKC scheme was implemented in Matlab as follows.

The beginning stage U_0 can be written as shown below, where u_n is the initial condition at the first time step, and after the first time step it would be the numerical approximation at that time.

$$U_0 = u_n \tag{17}$$

$$U_1 = U_0 = \tilde{\mu_1} k F_0 \tag{18}$$

Once these two stages are computed, the next step involves a loop to compute U_j for each stage. Let s represent the number of stages. Then the index j would range from 2 to s.

$$U_j = (1 - \mu_j - \nu_j)U_0 + \mu_j U_{j-1} + \nu_j U_{j-2} + \tilde{\mu} k F_{j-1} + \tilde{\gamma_j} k F_0 \tag{19}$$

The coefficients needed to compute U_j can be computed as follows.

$$\tilde{\mu_1} = b_1 \omega_1 \tag{20}$$

$$\mu_j = \frac{2 b_j \omega_0}{b_{j-1}}, \quad \nu_j = \frac{-b_j}{b_{j-2}}, \quad \tilde{\mu_j} = \frac{2 b_j \omega_1}{b_{j-1}}, \quad \tilde{\gamma}_{j-1}\tilde{\mu_j} \tag{21}$$

When we apply this method to the scalar stability test case $u'(t) = \lambda u(t)$, the relation $U_j = P_j(z)U_0$. Here, $z = \tau\lambda$ and $P_j(z)$ is a polynomial of degree j with $P_s(z)$ being the stability

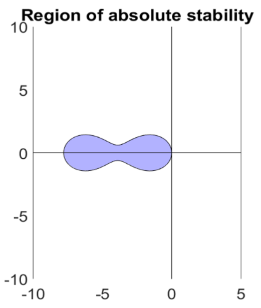


Figure 6: RKC method with 2 stages

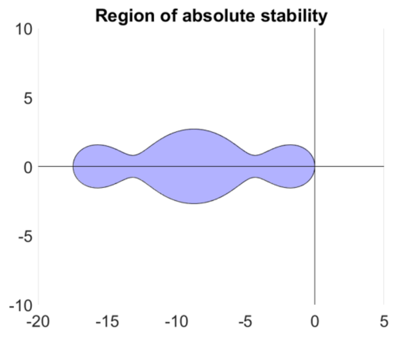


Figure 7: RKC method with 3 stages

polynomial. To satisfy the above relation, the Chebyshev three-term recursion condition must be met.

$$T_j(x) = 2xT_{j-1}(x) - T_{j-2}(x), \quad j = 2, 3, ..., s \tag{22}$$

where $T_0 = 1$ and $T_1(x) = x$

The "stability" polynomial $P_j(z)$ is then defined as the following.

$$P_j(z) = a_j + b_jT_j(\omega_0 + \omega_1 z), \quad a_j = 1 - b_jT_j(\omega_0) \tag{23}$$

where,

$$b_0 = b_2, \quad b_j = \frac{T''_j(\omega_0)}{(T'_j(\omega_0))^2}, \quad \omega_0 = 1 + \frac{\epsilon}{s^2}, \quad \omega_1 = \frac{T'_s(\omega_0)}{T''_s(\omega_0)}, \quad j = 2, 3, ..., s \tag{24}$$

6.2 Results

The results from the RKC method in time and central difference in space from the convergence study conducted in Matlab is shown in Figure 8. For this method k, the time step size is set as $k = 0.5h$ to avoid being right on the stability boundary of $k = h$. The results show that the method indeed is second order accurate.

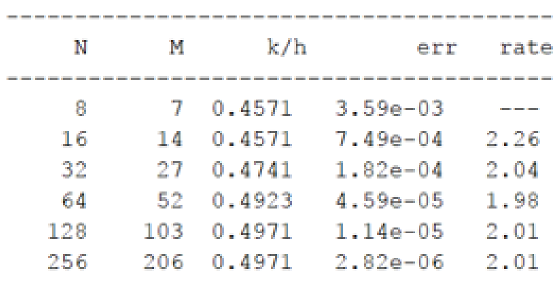
6.3 Pros and Cons

The advantage of Runge-Kutta compared to the Forward-Euler method is that we are able take much larger time steps. Thus, converging to a solution quicker. The Runge-Kutta Chebyshev also has the advantage to the Backward-Euler method in the sense that the RKC method does not involve an implicit solve. In more complex problems, this is a definite upside to the RKC method. In addition, the RKC method is second-order accurate while the Backward-Euler method is only first order accurate. The RKC method does require the calculation of the coefficients, but those only need to be done once. It also requires storing five stage vectors for the recursive relations. Overall, the pros outweigh the cons.

The stability region for the RKC method is dependent on the number of stages. If only one-stage is used, the one-stage RKC method is essentially the same as the Forward-Euler scheme and has the same region of stability. As we increase the number of stages, the region of stability also extends further along the negative-real axis. Figures 6 and 7 shows the region of stability for the RKC method using 2 and 3 stages respectively.

7 Conclusion

Each of the three methods discussed in this report have unique characteristics. The Forward-Euler scheme is an explicit second-order accurate method which requires many more time steps than the other two methods. It's region of stability is also much smaller. The Backward-Euler scheme has a much larger region of stability, making it more versatile. However, it is only first-order accurate and adds to computational cost by requiring an implicit solve at each time-step. The



```
----------------------------------------
     N        M       k/h          err    rate
----------------------------------------
     8        7   0.4571    3.59e-03     ---
    16       14   0.4571    7.49e-04    2.26
    32       27   0.4741    1.82e-04    2.04
    64       52   0.4923    4.59e-05    1.98
   128      103   0.4971    1.14e-05    2.01
   256      206   0.4971    2.82e-06    2.01
```

Figure 8: Runge-Kutta Chebyshev Matlab Convergence Study Results.

Runge-Kutta Chebyshev method has a region of stability along the negative real axis, meaning it has some restrictions on where it will be stable. It is a second-order accurate method and involves calculations of the parameters. In more complex problems, as the grid resolution needs to be finer, the RKC method would prove the most useful. The only restriction would be to ensure that the eigenvalues of the system strictly lie on the negative real axis [VSH04].

References

[VSH04] J.G. Verwer, B.P. Sommeijer, and W. Hundsdorfer. Rkc time-stepping for advection–diffusion–reaction problems. *Journal of Computational Physics*, 201(1):61 – 79, 2004.

```
% Gurpal Singh
% Math 567
% 12/13/2017

% About:
%   This project uses 3 different methods to approximate the solution
%   to diffusion in a one-dimensional rod. The methods used are:
%       i.   Explicit Method:           Forward Euler + Central
 Difference
%       ii.  Implicit/Direct Method:    Backward Euler + Central
 Difference
%       iii. Runge-Kutta + Chebychev:   Central Difference + RKC
method = 'rkc';

% RKC Settings
order = 2;

plot_soln = false;
T = .4; % Total Time
% m = 1;
x_start = 0; % Starting x
x_end = 1;   % Ending x


% Eigenmode solution
eigmode = @(x,t,m) exp(-m^2*pi^2*t).*sin(m*pi*x);

m = 1;
% Exact Solution
uexact = @(x,t) eigmode(x,t,m);


uxx = @(x,t) -(pi^2)*(m^2).*sin(m.*pi.*x)*exp(-(m^2)*(pi^2)*t);

% Set Spatial Scale and Convergence Study values
N0 = 8;
I = 0:5;
Nvec = N0*2.^I;

% Print to screen
l = double('-')*ones(1,40);
fprintf('%s\n',l);
fprintf('%6s %6s %8s %10s %6s\n','N','M','k/h','err','rate');
fprintf('%s\n',l);

% Plot Title Settings
if plot_soln
    close all
    switch method
        case 'fe'
            tstr = 'Forward Euler';
        case 'be'
```

```
            tstr = 'Backward Euler';
        case 'rkc'
            tstr = 'RKC Method';
    end
end


for i = 1:length(I) % Convergence Study Loop
    N = Nvec(i);    % Set Nvalue
%     N = 10;
    h = (x_end-x_start)/N; % Define h based off of N

    xe = linspace(x_start,x_end,N+1)'; % x coordinates
    x = xe(2:end-1);                   % Interior Coordinates

    % Build A matrix with Dirichlet conditions
    z = ones(N-1,1);
    A = spdiags([z -2*z z],-1:1,N-1,N-1);

    switch method % Pick which Method
        case 'fe'
            k_est = 0.45*h^2;
            M = round(T/k_est) + 1;
            k = T/M;
            B = eye(N-1) + (k/h^2)*A;
        case 'be'
            k_est = h;
            M = round(T/k_est) + 1;
            k = T/M;
            B_lhs = eye(N-1) - (k/h^2)*A;
        case 'rkc'

            % Set k to 0.5h
            k_est = 0.5*h;
            M = round(T/k_est) + 1;
            k = T/M;


            % Calculate Spectral Radius of A
            A = (1/(h^2)) .*A;
            eigvals = eig(A);
            spec_rad = max(abs(eigvals));

            % Calculate RKC Parameters
            params = rkc_params(order, k, spec_rad);
    end

    t = zeros(1,M+1); % Time Vector

    un = uexact(x,0);    % Interior values only

    % Prepare Figure for Plot
    if (plot_soln)
        clf
        clear a b
        ue = uexact(xe,0);
        b = plot(xe,ue,'black','linewidth',2);
        hold on;
        a = plot(xe,[0;un;0],'ro','linewidth',2,'markersize',10);
        lh = legend([a,b],{'Approximate solution','Exact solution'});
        title(tstr,'fontsize',18);
        axis([0 1 -1 1]);
        set(gca,'fontsize',16);
    end


    for n = 1:M % Time Loop
        t(n+1) = n*k; % Set Current Time
        tn = t(n+1);
        switch method % Select Method
            case 'fe'
                unp1 = B*un;
            case 'be'
                unp1 = B_lhs\un;
            case 'rkc'

                % Set wn to initial condition for 1st timestep
                if (n == 1)
                wn = uexact(x,0);
                end

                % Calculate W0 and W1
                W0 = wn;
                MuT_1 = params.MuT(2);
                F0 = A*W0;
                W1 = W0 + MuT_1 * k * F0;

                W_jm1 = W1;
                W_jm2 = W0;

                for j = 2:params.s
                    % Set Coefficients
                    Mu_j = params.Mu(j+1);
                    Nu_j = params.Nu(j+1);
                    MuT_j = params.MuT(j+1);
                    GammaT_j = params.GammaT(j+1);
                    c_j = params.c(j+1);

                    % Calculate F_jm1
                    F_jm1 = A*W_jm1;

                    % Calculate Current Stage
                    Wj = (1 - Mu_j - Nu_j)*W0 + Mu_j*W_jm1 +
  Nu_j*W_jm2 ...
                        + MuT_j*k*F_jm1 + GammaT_j*k*F0;

                    % Update
                    W_jm2 = W_jm1;
                    W_jm1 = Wj;
                end


                % Save W_s
                unp1 = Wj;
                wn = Wj;
        end

        ue = uexact(xe,t(n+1)); % Compute Exact Solution at that
 timestep

        % Plotting
        U = [0; unp1; 0]; % Add on Boundary Conditions
            if (plot_soln)
                set(a,'ydata',U);
                set(b,'ydata',ue);
            end
        pause(0.6)
        un = unp1;
    end

    % Compute error at end of time step
    err(i) = norm(ue(2:end-1)-unp1,Inf);
    if (i == 1)
        rate = '---';
    else
        r = log(err(i-1)/err(i))/log(2);
        rate = sprintf('%6.2f',r);
    end
    fprintf('%6d %6d %7.4f %10.2e %6s\n',N,M,k/h,err(i),rate);
end
```

```
----------------------------------------
     N      M      k/h         err   rate
----------------------------------------
     8      7   0.4571   3.59e-03    ---
    16     14   0.4571   7.49e-04   2.26
    32     27   0.4741   1.82e-04   2.04
    64     52   0.4923   4.59e-05   1.98
   128    103   0.4971   1.14e-05   2.01
   256    206   0.4971   2.82e-06   2.01
```

Published with MATLAB® R2017a

```
 function rkc_params = RKC_parameters(order,dt,sprad,s)

if (nargin < 4)
    if (order == 1)
        s = 1 + ceil(sqrt(dt*sprad/0.19)); %s = 1+ceil(sqrt(dt*4/
(dx^2*0.653)));
    else
        s = 1 + ceil(sqrt(dt*sprad/0.653)); %s = 1+ceil(sqrt(dt*4/
(dx^2*0.653)));
    end
end

rkc_params.s = s;
rkc_params.order = order;

% ----------------------------------------------
% Construct Chebyshev polynomials recursively
% ----------------------------------------------

if order == 1
    ep = 0.05;
else
    ep = 2/13;
end

% ----------------------------------------------
% Construct Chebyshev polynomials recursively
% ----------------------------------------------

s_idx = s+1; % for indexing matlab arrays
idx_0 = 1;
idx_1 = 2;
idx_2 = 3;
w0 = 1 + ep/s^2;


T(idx_0) = 1;
T(idx_1) = w0;
T(idx_2) = 2*w0^2-1;

dT(idx_0) = 0;
dT(idx_1) = 1;
dT(idx_2) = 4*w0;

dT2(idx_0) = 0;
dT2(idx_1) = 0;
dT2(idx_2) = 4;

for j = 3:s
    j_idx = j+1;
    T(j_idx) = 2*w0*T(j_idx-1)-T(j_idx-2);
    dT(j_idx) = 2*w0*dT(j_idx-1)-dT(j_idx-2) + 2*T(j_idx-1);
    dT2(j_idx) = 2*w0*dT2(j_idx-1)-dT2(j_idx-2) + 4*dT(j_idx-1);
end

% ---------------------------------------
% Get parameters for each order
% ---------------------------------------
if order == 1
    w1 = T(s_idx)/dT(s_idx);

    b = NaN(1,s+1);
    for j = 0:s,
        j_idx = j+1;
        b(j_idx) = 1./T(j_idx);
    end
    Mu(idx_0) = NaN;
    MuT(idx_0) = NaN;

    Mu(idx_1) = b(idx_1)*w1;
    MuT(idx_1) = w1/w0;

    for j = 2:s
        j_idx = j+1;
        Mu(j_idx) = 2*b(j_idx)*w0/b(j_idx-1);
        Nu(j_idx) = -b(j_idx)/b(j_idx-2);
        MuT(j_idx)= 2*b(j_idx)*w1/b(j_idx-1);
        GammaT(j_idx) = 0;
    end

    c = NaN(1,s+1);
    for j = 1:s
        j_idx = j+1;
        c(j_idx) = T(s_idx)/dT(s_idx)*dT(j_idx)/T(j_idx);
    end
    c(idx_0) = 0;
    c(s_idx) = 1;   % already 1, but still might be a good idea to set
 to 1
else

    arg = s*log(w0+sqrt(w0^2-1));
    w1 = sinh(arg)*(w0^2-1)/(cosh(arg)*s*sqrt(w0^2-1)-w0*sinh(arg));
    % w1 = dT(s_idx)/dT2(s_idx);

    b = NaN(1,s+1);
    for j = 2:s
        j_idx = j+1;
        b(j_idx) = dT2(j_idx)/dT(j_idx)^2;
    end
    b(idx_0) = b(idx_2);
    b(idx_1) = b(idx_2);

    Mu(idx_0) = NaN;
    MuT(idx_0) = NaN;
    MuT(idx_1) = b(idx_1)*w1;
    for j = 2:s
```

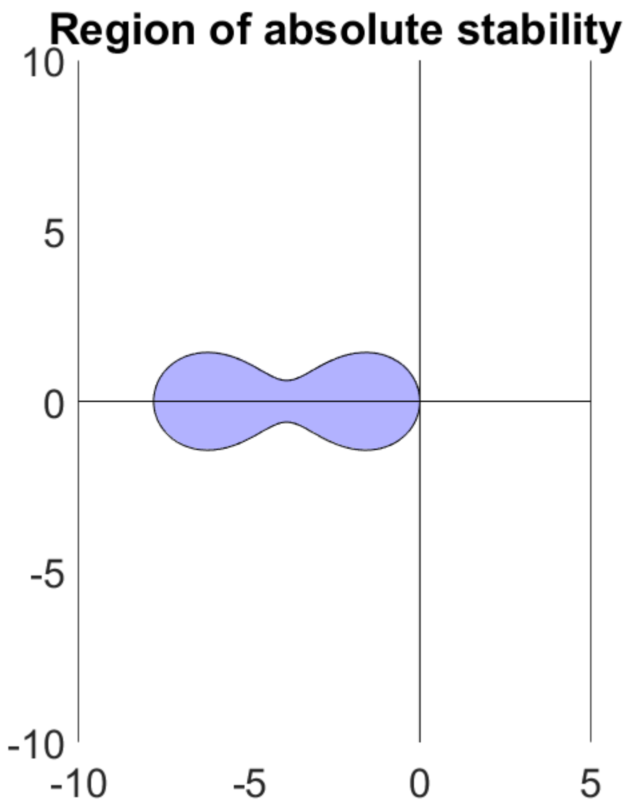
```
        j_idx = j+1;
        Mu(j_idx) = 2*b(j_idx)*w0/b(j_idx-1);
        Nu(j_idx) = -b(j_idx)/b(j_idx-2);
        MuT(j_idx)= 2*b(j_idx)*w1/b(j_idx-1);
        GammaT(j_idx) = -(1-b(j_idx-1)*T(j_idx-1))*MuT(j_idx);
    end

    c = NaN(1,s+1);
    for j = 2:s
        j_idx = j+1;
        c(j_idx) = dT(s_idx)/dT2(s_idx)*dT2(j_idx)/dT(j_idx);
    end
    c(idx_1) = c(idx_2)/dT(idx_2);   % needed here, but not for RKC1
    c(idx_0) = 0;
    c(s_idx) = 1;
end


% Store parameters
rkc_params.Mu = Mu;
rkc_params.Nu = Nu;
rkc_params.MuT = MuT;
rkc_params.GammaT = GammaT;
rkc_params.c = c;


end
```

Published with MATLAB® R2017a

```
% RKC Stability Region

% plotSrkc.m
%
% call plotS to plot the region of absolute stability S
% for a Runge-Kutta-Chebyshev method of order 1 or 2.
%
% Requires
%     chebpoly.m, chebpoly1.m, chebpoly2.m
%     plotS.m from chapter7.
%
% From  http://www.amath.washington.edu/~rjl/fdmbook/chapter8  (2007)


order = 1;              % order of accuracy (1 or 2)
r = 2;                  % number of stages
epsilon = 0.05;         % damping parameter
plotOS = true;

disp(' ' )
if plotOS
    disp(['Plotting S and OS for RKC method with s = ' num2str(r)])
  else
    disp(['Plotting S for RKC method with s = ' num2str(r)])
  end
disp(['  order = ' num2str(order)])
disp(['  epsilon = ' num2str(epsilon)])

if order==1
    % first-order Runge-Kutta-Chebyshev method:
    beta = 2*r^2;
    w0 = 1 + epsilon/r^2;
    w1 = chebpoly(r,w0) / chebpoly1(r,w0);
    R = @(z) chebpoly(r, w0 + w1*z) ./ chebpoly(r,w0);

  elseif order==2
    % second-order Runge-Kutta-Chebyshev method:
    w0 = 1 + epsilon/r^2;
    Trp = chebpoly1(r,w0);
    Trpp = chebpoly2(r,w0);
    w1 = Trp / Trpp;
    beta = (w0+1)*Trpp / Trp;
    br = Trpp / (Trp^2);
    ar = 1 - br*chebpoly(r,w0);
    R = @(z) ar + br*chebpoly(r, w0 + w1*z);

  else
    error('order must be 1 or 2')
  end

ylim = max(10, 2*r);
axisbox = [-beta-2 5 -ylim ylim];
```

```
% Now that R is set, plot S:
plotS(R,0,axisbox)
```

```
Plotting S and OS for RKC method with s = 2
  order = 1
  epsilon = 0.05
```



Published with MATLAB® R2017a

```
% Stability Region for Forward Euler
rho_FE = @(zeta) zeta - 1;
sigma_FE = @(zeta) 1;
z_FE = @(rho_AB4,sigma_AB4) rho_AB4./sigma_AB4;

theta = 0:.01:2*pi;
zeta = exp(i*theta);

rho_out = rho_FE(zeta);
sigma_out = sigma_FE(zeta);

z_out = z_FE(rho_out,sigma_out);

figure
whitebg('white')
plot(z_out)
fill(real(z_out),imag(z_out),'r')
hold on

title('Stability Region for Forward Euler')
xlabel('Re(z)')
ylabel('Im(z)')
grid
axis([-3 1 -2 2])


% Stability Region for Backward Euler
rho_BE = @(zeta) zeta - 1;
sigma_BE = @(zeta) -zeta;
z_BE = @(rho_AB4,sigma_AB4) rho_AB4./sigma_AB4;

rho_out = rho_BE(zeta);
sigma_out = sigma_BE(zeta);

z_out = z_BE(rho_out,sigma_out);


figure
rectangle('Position',[-3 -2 4 4],'FaceColor','red')
hold on
plot(z_out)
fill(real(z_out),imag(z_out),'white')
hold on

title('Stability Region for Backward Euler')
xlabel('Re(z)')
ylabel('Im(z)')
grid on
set(gca,'layer','top');
axis([-3 1 -2 2])
```

Published with MATLAB® R2017a