Lecture 5

Stiffness and Implicit Methods

5.1 Stiffness

Stiffness is a general (though somewhat fuzzy) term to describe systems of equations which exhibit phenoma at widely-varying scales. For the ODE's we have been studying, this means widely-varying timescales.

One way for stiffness to arise is through a difference in timescales between a forcing timescale and any characteristic timescales of the unforced system. For example, consider the following problem:

$$u_t + 1000u = 100 \sin t, \qquad u(0) = 1. \tag{5.1}$$

The forcing term oscillates with a frequency of 1. By comparison, the unforced problem decays very rapidly since the eigenvalue is $\lambda = -1000$. Thus, the timescales are different by a factor of 1000.

Suppose we are only interested in the long time behavior of $u(t)$, not the initial transient. We would like to take a timestep that would be set by the requirements to resolve the $\sin t$ forcing. For example, one might expect that setting $\Delta t = 2\pi/100$ (which would result in 100 timesteps per period of the forcing) would be sufficient to have reasonable accuracy. However, if the method does not have a large eigenvalue stability region, this may not be possible. If a forward Euler method is applied to this problem, eigenvalue stability would limit the $\Delta t \leq 0.002$ (since the eigenvalue is $\lambda = -1000$ and the forward Euler stability region crosses the real axis at -2). The results from simulations for a variety of Δt using forward Euler are shown in Figure 5.1. For $\Delta t = 0.001$, the solution is well behaved and looks realistic. For $\Delta t = 0.0019$, the approach of eigenvalue instability is evident as there are oscillations during the first few iterations which eventually decay. For $\Delta t = 0.002$, the oscillations no longer decay but remain throughout the entire simulation. Finally, for $\Delta t = 0.0021$, the oscillations grow unbounded. A zoomed image of these results concentrating on the initial time behavior is shown in Figure 5.2.

A more efficient approach to numerically integrating this stiff problem would be to use a method with eigenvalue stability for large negative real eigenvalues. Implicit methods often have excellent stability along the negative real axis. The simplest implicit method is the backward Euler method,

$$v^{n+1} = v^n + \Delta t f(v^{n+1}, t^{n+1}). \tag{5.2}$$

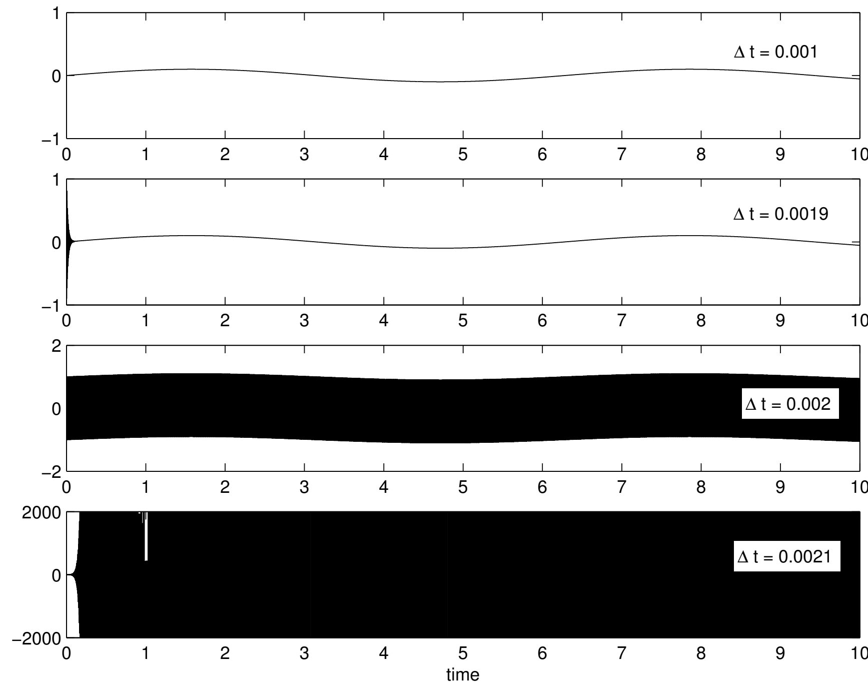


Figure 5.1: Forward Euler solution for $u_t + 1000u = 100 \sin t$ with $u(0) = 1$ at $\Delta t = 0.001$, 0.0019, 0.002, and 0.0021.

The backward Euler method is first order accurate ($p = 1$). The amplification factor for this method is,

$$g = \frac{1}{1 - \lambda \Delta t} \tag{5.3}$$

When λ is negative real, then $g < 1$ for all Δt. The eigenvalue stability region for the backward Euler method is shown in Figure 5.3. Only the small circular portion in the right-half plane is unstable while the entire left-half plane is stable. Results from the application of the backward Euler method to Equation 5.1 are shown in Figure 5.4. The excellent stability properties of this method are clearly seen as the solution looks acceptable for all of the tested Δt. Clearly, for the larger Δt, the initial transient is not accurately simulated, however, it does not effect the stability. Thus, as long as the initial transient is not desired, the backward Euler method will likely be a more effective solution strategy than the forward Euler method for this problem.

Another popular implicit method is trapezoidal integration,

$$v^{n+1} = v^n + \frac{1}{2}\Delta t \left[f(v^{n+1}, t^{n+1}) + f(v^n, t^n) \right]. \tag{5.4}$$

Trapezoidal integration is second-order accurate ($p = 2$). The amplification factor is,

$$g = \frac{1 + \frac{1}{2}\lambda \Delta t}{1 - \frac{1}{2}\lambda \Delta t}. \tag{5.5}$$

The stability boundary for trapezoidal integration lies on the imaginary axis (see Figure 5.5). Again, this method is stable for the entire left-half plane thus it will work well for stiff

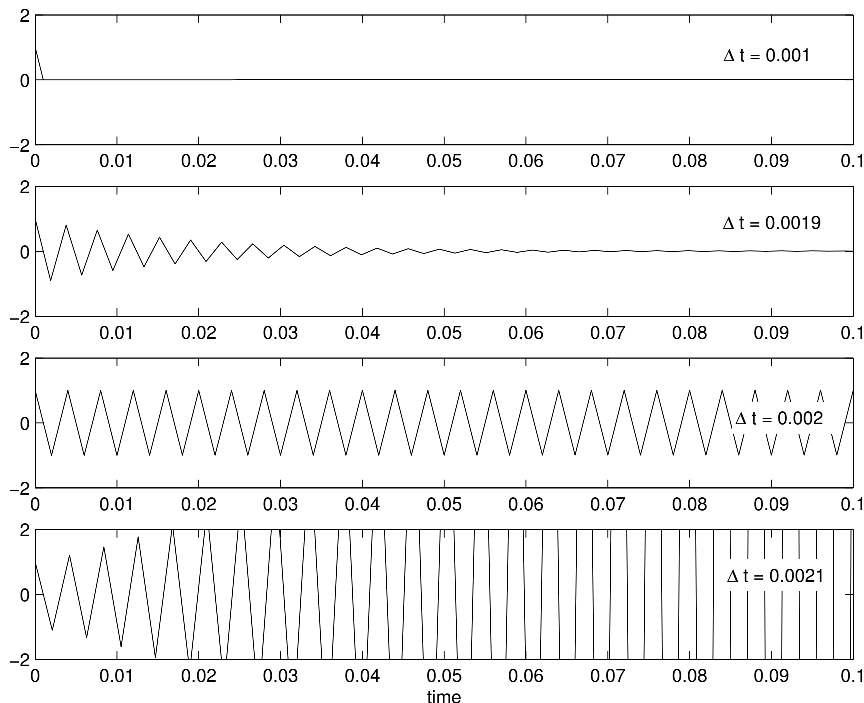


Figure 5.2: Forward Euler solution for $u_t + 1000u = 100 \sin t$ with $u(0) = 1$ at $\Delta t = 0.001$, 0.0019, 0.002, and 0.0021. Same results as in Figure 5.1 just showing the small t behavior in more detail.

problems.

The accuracy of the forward Euler, backward Euler, and trapezoidal integration methods are compared in Figure 5.6 for Equation 5.1. The error is computed as the maximum across all timesteps of the difference between numerical and exact solutions,

$$E = \max_{n=[0,T/\Delta t]} |v^n - u(n\Delta t)|.$$

These results show that the forward Euler method is first order accurate (since the slope on the log-log scaling is 1) once the Δt is small enough to have eigenvalue stability (for $\Delta t > 0.002$ the algorithm is unstable and the errors are essentially unbounded). In contrast, the two implicit methods have reasonable errors for all Δt's. As the Δt become small, the slope of the backward Euler and trapezoidal methods become essentially 1 and 2 (indicating first and second order accuracy). Clearly, if high accuracy is required, the trapezoidal method will require fewer timesteps to achieve this accuracy.

Stiffness can also arise in linear or linearized systems when eigenvalues exist with significantly different magnitudes. For example,

$$u_t = Au, \qquad A = \begin{pmatrix} -1 & 1 \\ 0 & -1000 \end{pmatrix}.$$

The eigenvalues of A are $\lambda = -1$ and $\lambda = -1000$. Since the timestep must be set so that both eigenvalues are stable, the larger eigenvalue will dominate the timestep. The spectral

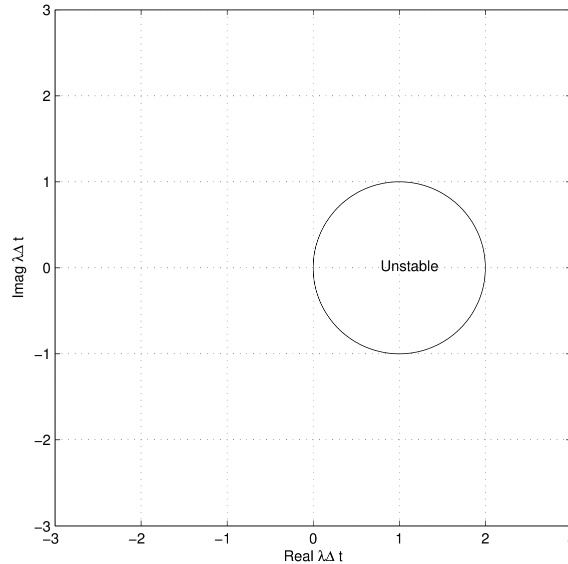


Figure 5.3: Backward Euler stability region

condition number is the ratio of the largest to smallest eigenvalue magnitudes,

$$\text{Spectral Condition Number} = \frac{\max|\lambda_j|}{\min|\lambda_j|}$$

When the spectral condition number is greater than around 1000, problems are starting to become stiff and implicit methods are likely to be more efficient than explicit methods.

Example 5.1 (Stiffness from PDE discretizations) *One of the more common ways for a stiff system of ODE's to arise is in the discretization of time-dependent partial differential equations (PDE's). For example, consider a one-dimensional heat diffusion problem that is modeled by the following PDE for the temperature, T:*

$$T_t = \frac{k}{\rho c_p} T_{xx}.$$

where ρ, c_p, and k are the density, specific heat, and thermal conductivity of the material, respectively. Suppose the physical domain for of length L from $x = 0$ to $x = L$. A finite difference approximation in x might divide the physical domain into a set of equally spaced nodes with distance $h = L/(N-1)$ where N is the total number of nodes including the endpoints. So, node i would be located at $x_i = ih$. Then, at each node, T_{xx} is approximated using a finite difference derivative. For example, at node i we might use the following approximation,

$$T_{xx}\big|_i = \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2}.$$

Note: we will discuss finite difference discretizations of PDE's in detail in later lectures. Using this in the heat diffusion equation, we can find the time rate of change at node i as,

$$T_t\big|_i = \frac{k}{\rho c_p} \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2}.$$

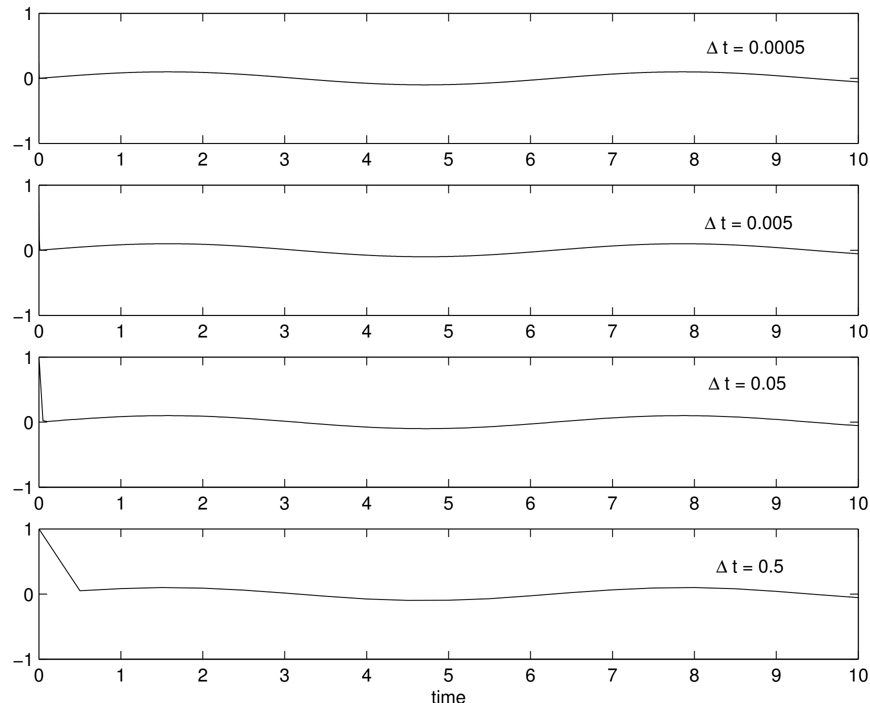


Figure 5.4: Backward Euler solution for $u_t + 1000u = 100 \sin t$ with $u(0) = 1$ at $\Delta t = 0.0005$, 0.005, 0.05, and 0.5.

Thus, the T_t at node i depends on the values of T at nodes $i-1$, i, and $i+1$ in a linear manner. Since each node in the interior of the domain will satisfy the same equation, we can put the finite difference discretization of the heat diffusion problem into our standard system of ODE's form,

$$u_t = Au + b, \qquad u = [T_2, T_3, T_4, \ldots, T_N]^T,$$

where A will be a tri-diagonal matrix (i.e. only the main diagonal and the two neighboring diagonals will be non-zero) since the finite difference approximation only depends on the neighboring nodes. The vector b will depend on the specific boundary conditions.

The question is how do the eigenvalues of A behave, in particular, as the node spacing h is decreased. To look at this, we arbitrarily choose,

$$\frac{k}{\rho c_p} = 1 \qquad L = 1$$

since the magnitudes of these parameters will scale the magnitude of the eigenvalues of A by the same value but not alter the ratio of eigenvalues (the ratio is only altered by the choice of h/L). Figure 5.7 shows the locations of the eigenvalues for $h/L = 0.1$ and $h/L = 0.05$. The eigenvalues are negative real numbers. The smallest magnitude eigenvalues appear to be nearly unchanged by the different values of h. However, the largest magnitude eigenvalues appear to have increased by a factor of 4 from approximately -400 to -1600 when h decreased by a factor of 2. This suggests that the ratio of largest-to-smallest magnitude eigenvalues (i.e. the spectral condition number) is $O(1/h^2)$. Table 5.1 confirms this depends for a range of

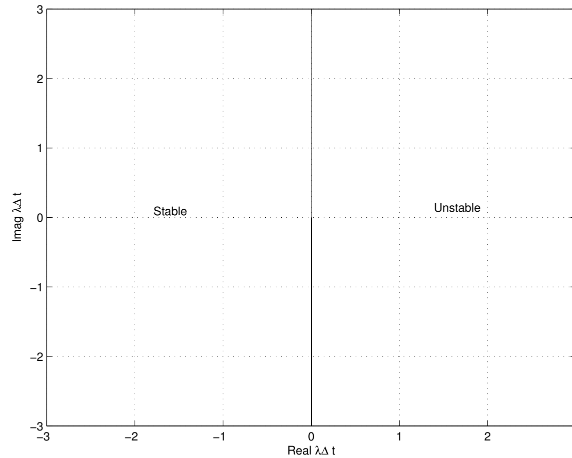


Figure 5.5: Trapezoidal intgeration stability region

h/L	$\min\lvert\lambda\rvert$	$\max\lvert\lambda\rvert$	$\max\lvert\lambda\rvert/\min\lvert\lambda\rvert$
0.001	9.87	3999990	405284
0.01	9.86	39990	4052
0.1	9.79	390	40

Table 5.1: Minimum and maximum magnitude eigenvalues for one-dimensional diffusion

h/L values and also confirms that the smallest eigenvalue changes very little as h decreases.

In-class Discussion 5.1 (How does Δt vary with h for forward Euler?)

5.2 Implicit Methods for Linear Systems of ODE's

While implicit methods can allow significantly larger timesteps, they do involve more work than explicit methods. Consider the forward method applied to $u_t = Au$ where A is a $d \times d$ matrix.

$$v^{n+1} = v^n + \Delta t A v^n.$$

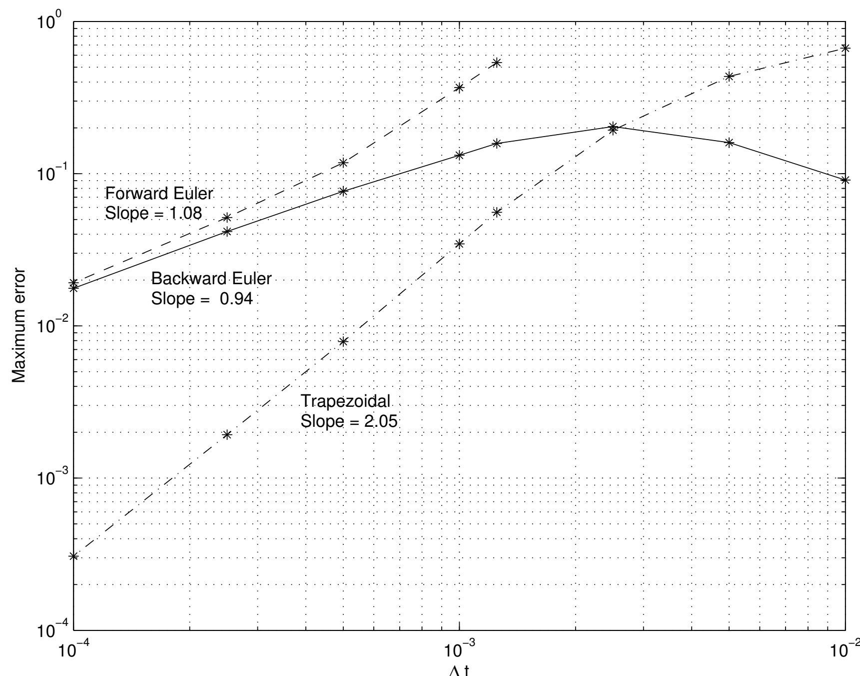


Figure 5.6: Comparison of error for forward Euler, backward Euler, and trapezoidal integration versus Δt for $u_t + 1000u = 100 \sin t$ with $u(0) = 1$.

In this explicit algorithm, the largest computational cost is the matrix vector multiply, Av^n which is an $O(d^2)$ operation. Now, for backward Euler,

$$v^{n+1} = v^n + \Delta t A v^{n+1}.$$

Re-arranging to solve for v^{n+1} gives:

$$\begin{aligned} v^{n+1} &= v^n + \Delta t A v^{n+1}, \\ v^{n+1} - \Delta t A v^{n+1} &= v^n, \\ (I - \Delta t A)\, v^{n+1} &= v^n, \end{aligned}$$

Thus, to find v^{n+1} requires the solution of a $d \times d$ system of equations which is an $O(d^3)$ cost. As a result, for large systems, the cost of the $O(d^3)$ linear solution may begin to outweigh the benefits of the larger timesteps that are possible when using implicit methods.

5.3 Implicit Methods for Nonlinear Problems

When the ODE's are nonlinear, implicit methods require the solution of a nonlinear system of algebraic equations at each iteration. To see this, consider the use of the trapezoidal method for a nonlinear problem,

$$v^{n+1} = v^n + \frac{1}{2}\Delta t \left[f(v^{n+1}, t^{n+1}) + f(v^n, t^n) \right].$$

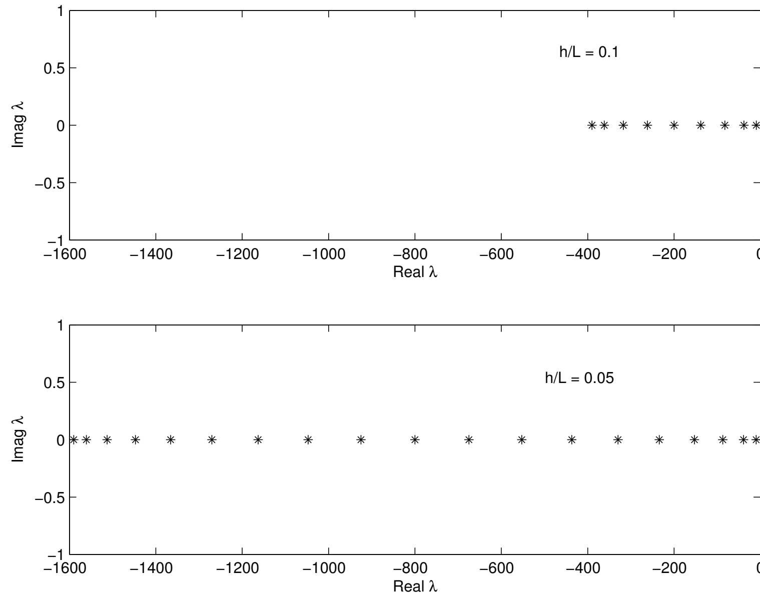


Figure 5.7: Eigenvalues for discretization of one-dimenionsal diffusion equation for $h/L = 0.1$ and $h/L = 0.05$.

We can define the following residual vector for the trapezoidal method,

$$R(w) \equiv w - v^n - \frac{1}{2}\Delta t\left[f(w, t^{n+1}) + f(v^n, t^n)\right].$$

Thus, v^{n+1} for the trapezoidal method is given by the solution of,

$$R(v^{n+1}) = 0,$$

which is a nonlinear algebraic system of equations for v^{n+1}.

One of the standard methods for solving a nonlinear system of algebraic equations is the Newton-Raphson method. It begins with an initial guess for v^{n+1} and solves a linearized version of $R = 0$ to find a correction to the initial guess for v^{n+1}. So, define the current guess for v^{n+1} as w^m where m indicates the sub-iteration in the Newton-Raphson method. Note, common useage is to call the iterations in the Newton-Raphson solution for v^{n+1} sub-iterations since these are iterations which occur within every time iteration from n to $n+1$. To find the correction, Δw, where

$$w^{m+1} = w^m + \Delta w,$$

we linearize and solve the nonlinear residual equation,

$$\begin{aligned} R(w^{m+1}) &= 0, \\ R(w^m + \Delta w) &= 0, \\ R(w^m) + \left.\frac{\partial R}{\partial w}\right|_{w^m} \Delta w &= 0, \\ \left.\frac{\partial R}{\partial w}\right|_{w^m} \Delta w &= -R(w^m). \end{aligned} \tag{5.6}$$

p	α_1	α_2	α_3	α_4	β_0
1	-1				1
2	$-\frac{4}{3}$	$\frac{1}{3}$			$\frac{2}{3}$
3	$-\frac{18}{11}$	$\frac{9}{11}$	$-\frac{2}{11}$		$\frac{6}{11}$
4	$-\frac{48}{25}$	$\frac{36}{25}$	$-\frac{16}{25}$	$\frac{3}{25}$	$\frac{12}{25}$

Table 5.2: Coefficients for backward differentiation methods

This last line is a linear system of equations for the correction since $\partial R/\partial w$ is a $d \times d$ matrix when the original ODE's are a system of d equations. For example, for the trapezoidal method,

$$\left.\frac{\partial R}{\partial w}\right|_{w^m} = I - \frac{1}{2}\Delta t \left.\frac{\partial f}{\partial w}\right|_{w^m}.$$

Usually, the initial guess for v^{n+1} is the previous iteration, i.e. $w^0 = v^n$. So, the entire iteration from n to $n+1$ has the following form,

1. Set initial guess: $w^0 = v^n$ and $m = 0$.
2. Calculate residual $R(w^m)$ and linearization $\partial R/\partial w|_{w^m}$.
3. Solve Equation 5.6 for Δw.
4. Update $w^{m+1} = w^m + \Delta w$.
5. Check if $R(w^{m+1})$ is small. If not, perform another sub-iteration.

5.4 Backwards Differentiation Methods

Backwards differentiation methods are one of the best multi-step methods for stiff problems. The backwards differentiation formulae are of the form,

$$v^{n+1} + \sum_{i=1}^{s} v^{n+1-i} = \Delta t \beta_0 f^{n+1}. \tag{5.7}$$

The coefficients for the first through fourth order methods are given in Table 5.2. The stability boundary for these methods are shown in Figure 5.8. As can be seen, all of these methods are stable everywhere on the negative real axis, and are mostly stable in the left-half plane in general. Thus, backwards differentiation work well for stiff problems in which stong damping is present.

Example 5.2 (Matlab's ODE Integrators) *Matlab has a a set of tools for integration of ODE's. We will briefly look at two of them:* ***ode45*** *and* ***ode15s****.* ***ode45*** *is designed to solve problems that are not stiff while* ***ode15s*** *is intended for stiff problems.* ***ode45*** *is based on a four and five-stage Runge-Kutta integration (discussed in Lecture 6), while* ***ode15s*** *is based*

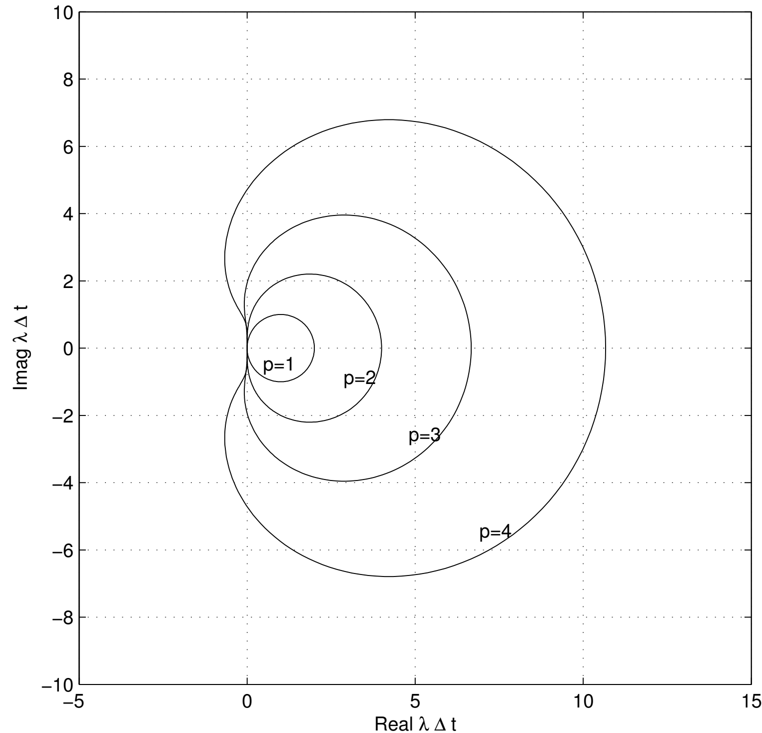


Figure 5.8: Backwards differentiation stability regions for $p = 1$ through $p = 4$ method. Note: interior of curves is unstable region.

*on a range of highly stable implicit integration formulas (one option when using **ode15s** is to use the backwards differentiation formulas). As a short illustration on how these Matlab ODE integrators are implemented, the following script solves the one-dimensional diffusion problem from Example 5.1 using either **ode45** or **ode15s**. The specific problem we consider here is a bar which is initially at a temperature $T_{init} = 400K$ and at $t = 0$, the temperature at the left and right ends is suddenly raised to $800K$ and $1000K$, respectively.*

```
% Matlab script: dif1d_main.m
%
% This code solve the one-dimensional heat diffusion equation
% for the problem of a bar which is initially at T=Tinit and
% suddenly the temperatures at the left and right change to
% Tleft and Tright.
%
% Upon discretization in space by a finite difference method,
% the result is a system of ODE's of the form,
%
% u_t = Au + b
%
% The code calculates A and b.  Then, uses one of Matlab's
% ODE integrators, either ode45 (which is based on a Runge-Kutta
```

```
% method and is not designed for stiff problems) or ode15s (which
% is based on an implicit method and is designed for stiff problems).
%

clear all; close all;

sflag = input('Use stiff integrator? (1=yes, [default=no]): ');

% Set non-dimensional thermal coefficient
k   = 1.0; % this is really k/(rho*cp)

% Set length of bar
L = 1.0; % non-dimensional

% Set initial temperature
Tinit = 400;

% Set left and right temperatures for t>0
Tleft = 800;
Tright = 1000;

% Set up grid size
Nx = input(['Enter number of divisions in x-direction: [default=' ...
            '51]']);
if (isempty(Nx)),
  Nx = 51;
end

h  = L/Nx;
x  = linspace(0,L,Nx+1);

% Calculate number of iterations (Nmax) needed to iterate to t=Tmax
Tmax = 0.5;

% Initialize a sparse matrix to hold stiffness & identity matrix
A = spalloc(Nx-1,Nx-1,3*(Nx-1));
I = speye(Nx-1);

% Calculate stiffness matrix

for ii = 1:Nx-1,

  if (ii > 1),
    A(ii,ii-1) = k/h^2;
```

```
  end

  if (ii < Nx-1),
    A(ii,ii+1) = k/h^2;
  end

  A(ii,ii) = -2*k/h^2;

end

% Set forcing vector
b = zeros(Nx-1,1);
b(1)    = k*Tleft/h^2;
b(Nx-1) = k*Tright/h^2;

% Set initial vector
v0 = Tinit*ones(1,Nx-1);

if (sflag == 1),

  % Call ODE15s
  options = odeset('Jacobian',A);
  [t,v] = ode15s(@dif1d_fun,[0 Tmax],v0,options,A,b);

else

  % Call ODE45
  [t,v] = ode45(@dif1d_fun,[0 Tmax],v0,[],A,b);

end

% Get midpoint value of T and plot vs. time
Tmid = v(:,floor(Nx/2));
plot(t,Tmid);
xlabel('t');
ylabel('T at midpoint');
```

As can be seen, this script pre-computes the linear system A and the column vector b since the forcing function for the one-dimensional diffusion problem can be written as the linear function, $f = Av + b$. Then, when calling either ODE integrator, the function which returns f is the first argument in the call and is named, ***dif1d_fun****. This function is given below:*

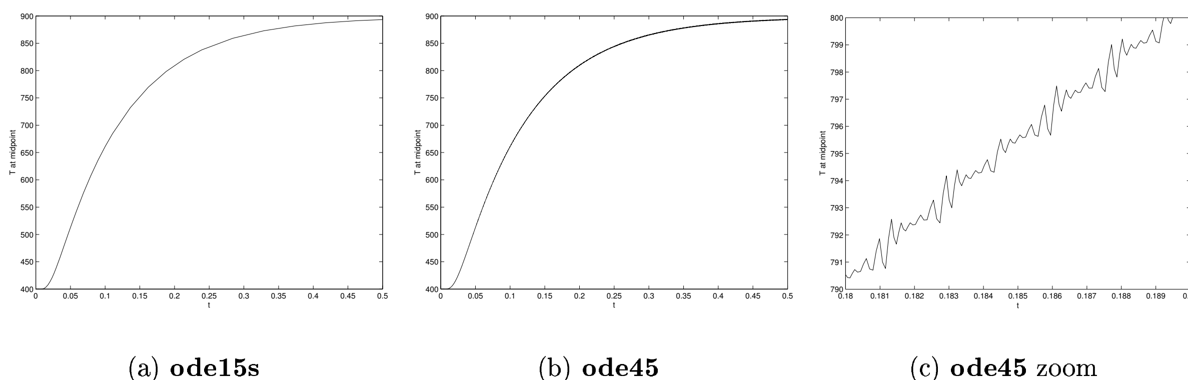
```
% Matlab function: dif1d_fun.m
%
```

```
% This routine returns the forcing term for
% a one-dimensional heat diffusion problem
% that has been discretized by finite differences.
% Note that the matrix A and the vector b are pre-computed
% in the main driver routine, dif1d_main.m, and passed
% to this function.  Then, this function simply returns
% f(v) = A*v + b.  So, in reality, this function is
% not specific to 1-d diffusion.

function [f] = dif1d_fun(t, v, A, b)

f = A*v + b;
```

*As can be seen from **dif1d_fun**, A and b have been passed into the function and thus the calculation of f simply requires the multiplication of v by A and the addition of b.*

The major difference between the implementation of the ODE integrators in Matlab and our discussions is that Matlab's implementations are adaptive. Specifically, Matlab's integrators estimate the error at each iteration and then adjust the timestep to either improve the accuracy (i.e. by decreasing the timestep) or efficiency (i.e. by increasing the timestep).

*The results for the stiff integrator, **ode15s** are shown in Figure 5.9(a). These results*



(a) **ode15s** (b) **ode45** (c) **ode45** zoom

Figure 5.9: Temperature evolution at the middle of bar with suddenly raised end temperatures using Matlab's **ode15s** and **ode45** integrators.

look as expected (note: in integrating from $t = 0$ to $t = 0.5$, a total of 64 timesteps were taken).

*The results for the non-stiff integrator are shown in Figure 5.9(b) and in a zoomed view in Figure 5.9(c). The presence of small scale oscillations can be clearly observed in the **ode45** results. These oscillations are a result of the large negative eigenvalues which require small Δt to maintain stability. Since the **ode45** method is adaptive, the timestep automatically decreases to maintain stability, but the oscillatory results clearly show that the stability is barely achieved. Also, as a measure of the relative inefficiency of the **ode45** integrator for this stiff problem, note that 6273 timesteps were required to integrate from $t = 0$ to $t = 0.5$.*

*One final concern regarding the efficiency of the stiff integrator **ode15s**. In order for this method to work in an efficient manner for large systems of equations such as in this example, it is very important that the Jacobian matrix, $\partial f/\partial u$ be provided to Matlab. If this is not done, then **ode15s** will construct an approximation to this derivative matrix using finite differences and for large systems, this will become a significant cost. In the script **dif1d_main**, the Jacobian is communicated to the **ode15s** integrator using the **odeset** routine. Note: **ode45** is an explicit method and does not need the Jacobian so it is not provided in that case.*