Chapter 9
Implicit Methods for Linear and Nonlinear Systems of ODEs

In the previous chapter, we investigated stiffness in ODEs. Recall that an ODE is stiff if it exhibits behavior on widely-varying timescales. Our primary concern with these types of problems is the eigenvalue stability of the resulting numerical integration method. Implicit methods offer excellent eigenvalue stability properties for stiff systems.

33 Self-Assessment

Before reading this chapter, you may wish to review...

- Linear systems of equations [18.02 Lecture 4]

After reading this chapter you should be able to...

- implement an implicit method for linear and nonlinear systems of ODEs
- describe and implement the Newton-Raphson method for solving nonlinear systems of algebraic equations
- identify and implement a backwards differentiation method
- discuss the Matlab suite of tools for numerical integration of ODEs

34 Implicit methods for linear systems of ODEs

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

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

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.

35 Implicit Methods for Nonlinear Problems

When the ODEs 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,

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

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,

$$\begin{aligned} R(w^{m+1}) &= 0, \\ R(w^m + \Delta w) &= 0, \\ R(w^m) + \left.\frac{\partial R}{\partial w}\right|_{w^m} \Delta w &= 0, \\ \left.\frac{\partial R}{\partial w}\right|_{w^m} \Delta w &= -R(w^m). \end{aligned} \tag{73}$$

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 ODEs are a system of d equations. For example, for the trapezoidal method,

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

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 73 for Δw.
4. Update $w^{m+1} = w^m + \Delta w$.
5. Check if $R(w^{m+1})$ is small. If not, set $m \leftarrow m+1$ and repeat steps 1 through 4.

Exercise 1. Implement a Matlab function that uses Newton-Raphson iteration to find a zero of the function $f(x) = x^3 - 15x^2 + 30$. The function should take as input a starting point `x0` and return as output the first zero it finds. (Note that it may not find the zero nearest the starting point.) You should use as stopping criterion that the function value at the last iteration is below 10^{-5} in absolute value.

36 Backwards Differentiation Methods

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

$$v^{n+1} + \sum_{i=1}^{s} \alpha_i v^{n+1-i} = \Delta t \beta_0 f^{n+1}. \tag{74}$$

The coefficients for the first through fourth order methods are given in Table 3. The stability boundary for these

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 3 Coefficients for backward differentiation methods

methods are shown in Figure 36. 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.

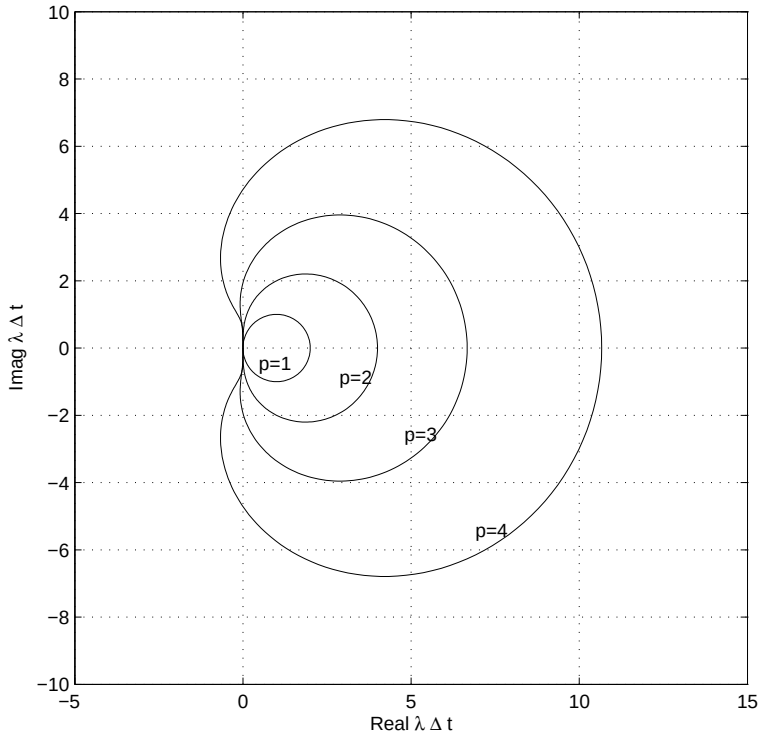


Fig. 12 Backwards differentiation stability regions for $p = 1$ through $p = 4$ method. Note: interior of curves is unstable region.

Exercise 2. Suppose we wish to simulate the ODE $u_t = Au$ defined by the 2-by-2 matrix A for $\Delta t = 0.1$. For which of the following A matrices will the fourth-order accurate backwards differentiation method be eigenvalue stable?

(a) $A = \begin{bmatrix} 0 & 1 \\ -180 & 0 \end{bmatrix}$

(b) $A = \begin{bmatrix} 0 & 1 \\ -290 & -2 \end{bmatrix}$

(c) $A = \begin{bmatrix} 0 & 1 \\ -8 & -4 \end{bmatrix}$

(d) $A = \begin{bmatrix} -5 & 0 \\ 0 & 5 \end{bmatrix}$

Example 1 (Matlab's ODE Integrators).

Matlab has 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 10), while **ode15s** is based 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 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;
```

```
35
36 % Set up grid size
37 Nx = input(['Enter number of divisions in x-direction: [default=' ...
38              '51]']);
39 if (isempty(Nx)),
40   Nx = 51;
41 end
42
43 h  = L/Nx;
44 x  = linspace(0,L,Nx+1);
45
46 % Calculate number of iterations (Nmax) needed to iterate to t=Tmax
47 Tmax = 0.5;
48
49 % Initialize a sparse matrix to hold stiffness & identity matrix
50 A = spalloc(Nx-1,Nx-1,3*(Nx-1));
51 I = speye(Nx-1);
52
53 % Calculate stiffness matrix
54
55 for ii = 1:Nx-1,
56
57   if (ii > 1),
58     A(ii,ii-1) = k/h^2;
59   end
60
61   if (ii < Nx-1),
62     A(ii,ii+1) = k/h^2;
63   end
64
65   A(ii,ii) = -2*k/h^2;
66
67 end
68
69 % Set forcing vector
70 b = zeros(Nx-1,1);
71 b(1)    = k*Tleft/h^2;
72 b(Nx-1) = k*Tright/h^2;
73
74 % Set initial vector
75 v0 = Tinit*ones(1,Nx-1);
76
77 if (sflag == 1),
78
79   % Call ODE15s
80   options = odeset('Jacobian',A);
81   [t,v] = ode15s(@dif1d_fun,[0 Tmax],v0,options,A,b);
82
83 else
84
85   % Call ODE45
86   [t,v] = ode45(@dif1d_fun,[0 Tmax],v0,[],A,b);
87
88 end
89
90 % Get midpoint value of T and plot vs. time
91 Tmid = v(:,floor(Nx/2));
92 plot(t,Tmid);
93 xlabel('t');
94 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;
```

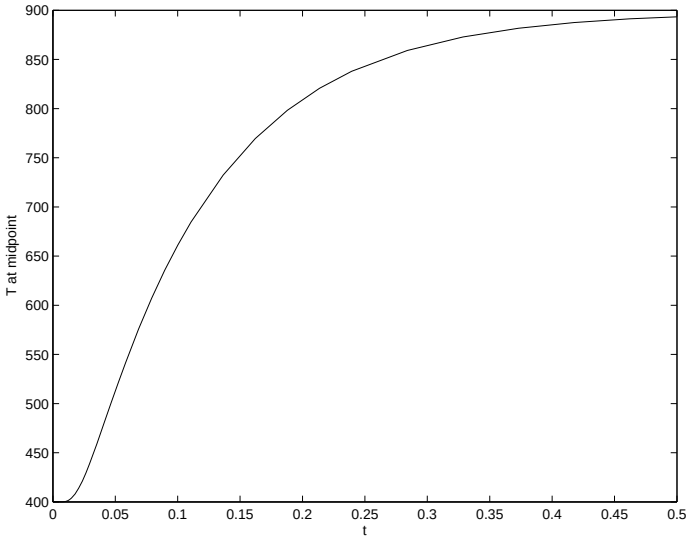
As can be seen from this function, A and b have been passed into the function and thus the calculation of f simply requires the multiplication of v by A and the addition of b.

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).

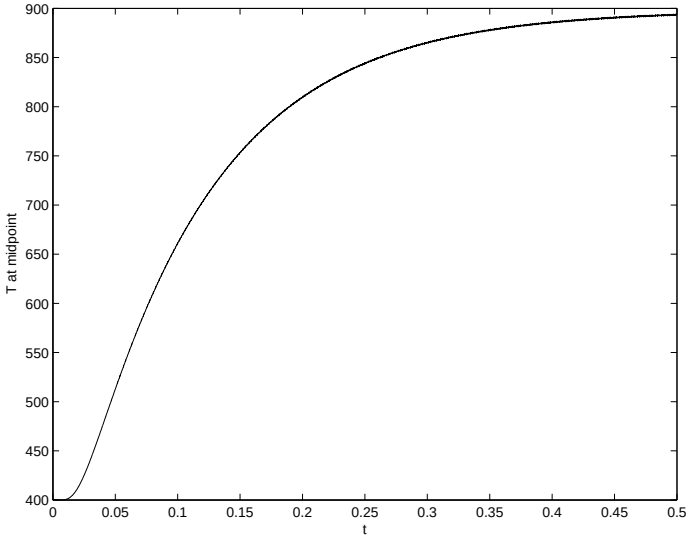
The results for the stiff integrator, **ode15s** are shown in Figure 13(a). These results look as expected (note: in integrating from $t = 0$ to $t = 0.5$, a total of 64 timesteps were taken).

The results for the non-stiff integrator are shown in Figure 13(b) and in a zoomed view in Figure 13(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 $t = 0.5$.

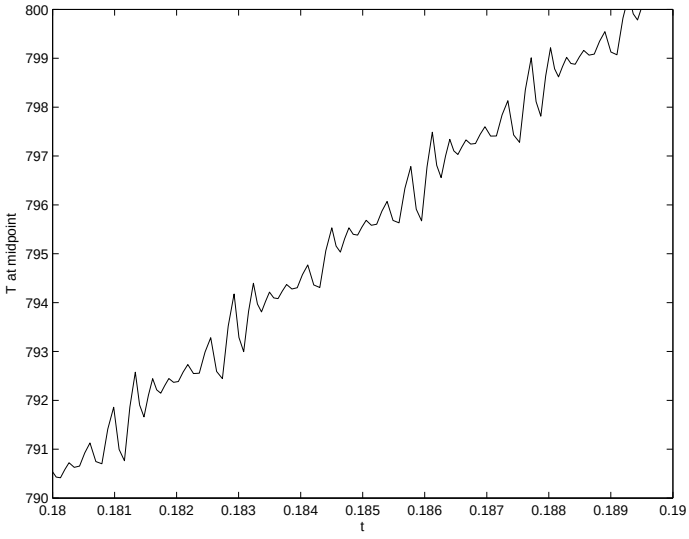
One final concern regarding the efficiency of the stiff integrator **ode15s**. In order for this method to work in an efficient manner for large systems of equations such as in this example, it is very important that the Jacobian matrix, $\partial f/\partial u$ be provided to Matlab. If this is not done, 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 main script, 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.



(a) **ode15s**



(b) **ode45**



(c) **ode45** zoom

Fig. 13 Temperature evolution at the middle of bar with suddenly raised end temperatures using Matlab's **ode15s** and **ode45** integrators.