

# Numerical Methods for Evolutionary Systems, Lecture 2

C. W. Gear

Celaya, Mexico, January 2007

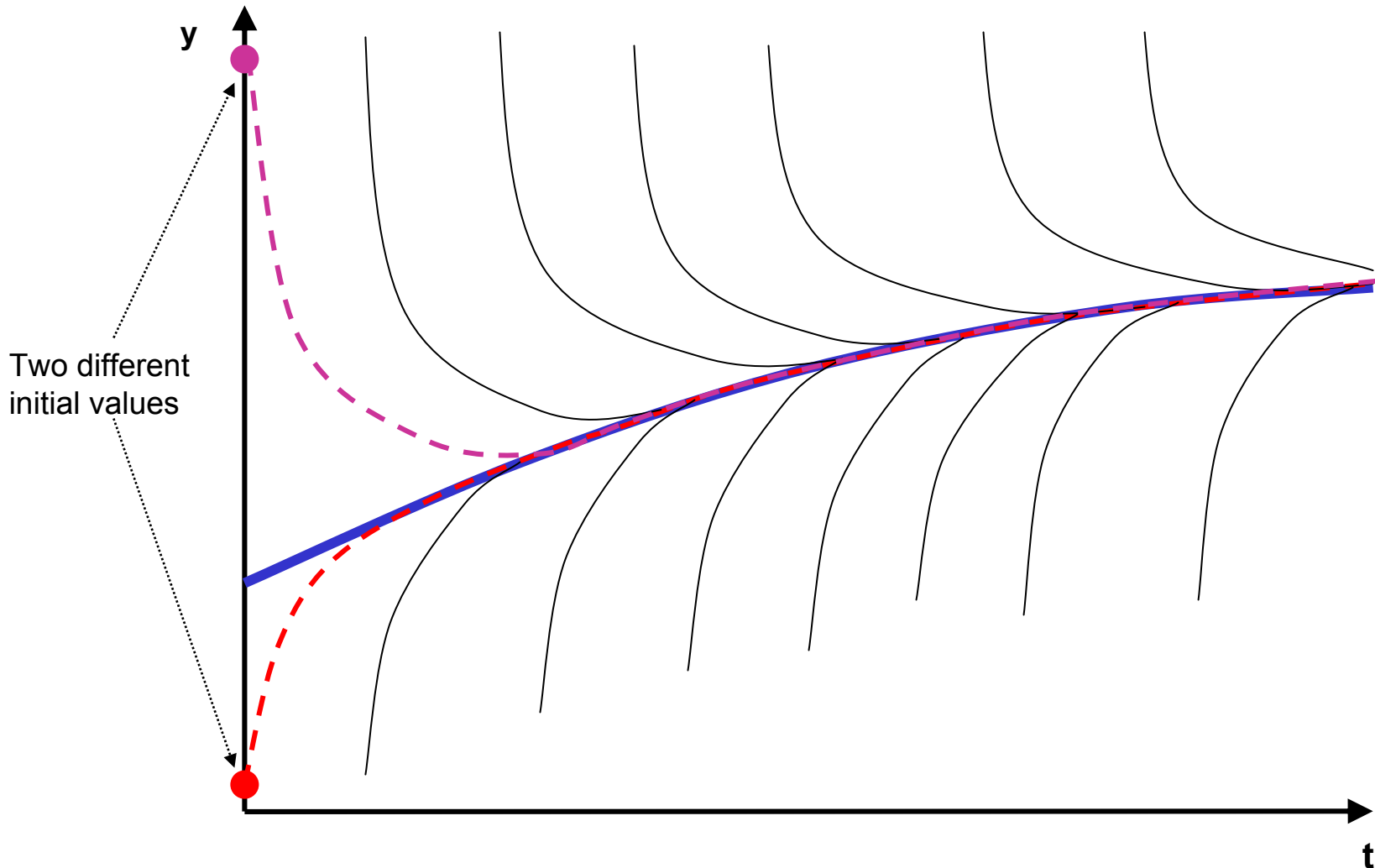
## Stiff Equations

The history of stiff differential equations goes back 55 years to the very early days of machine computation. The first identification of stiff equations as a special class of problems seems to have been due to chemists [chemical engineers] in 1952 (**C.F. Curtiss and J.O. Hirschfelder, *Integration of stiff equations*, Proc. of the National Academy of Sciences of U.S., 38 (1952), pp 235--243.)**)

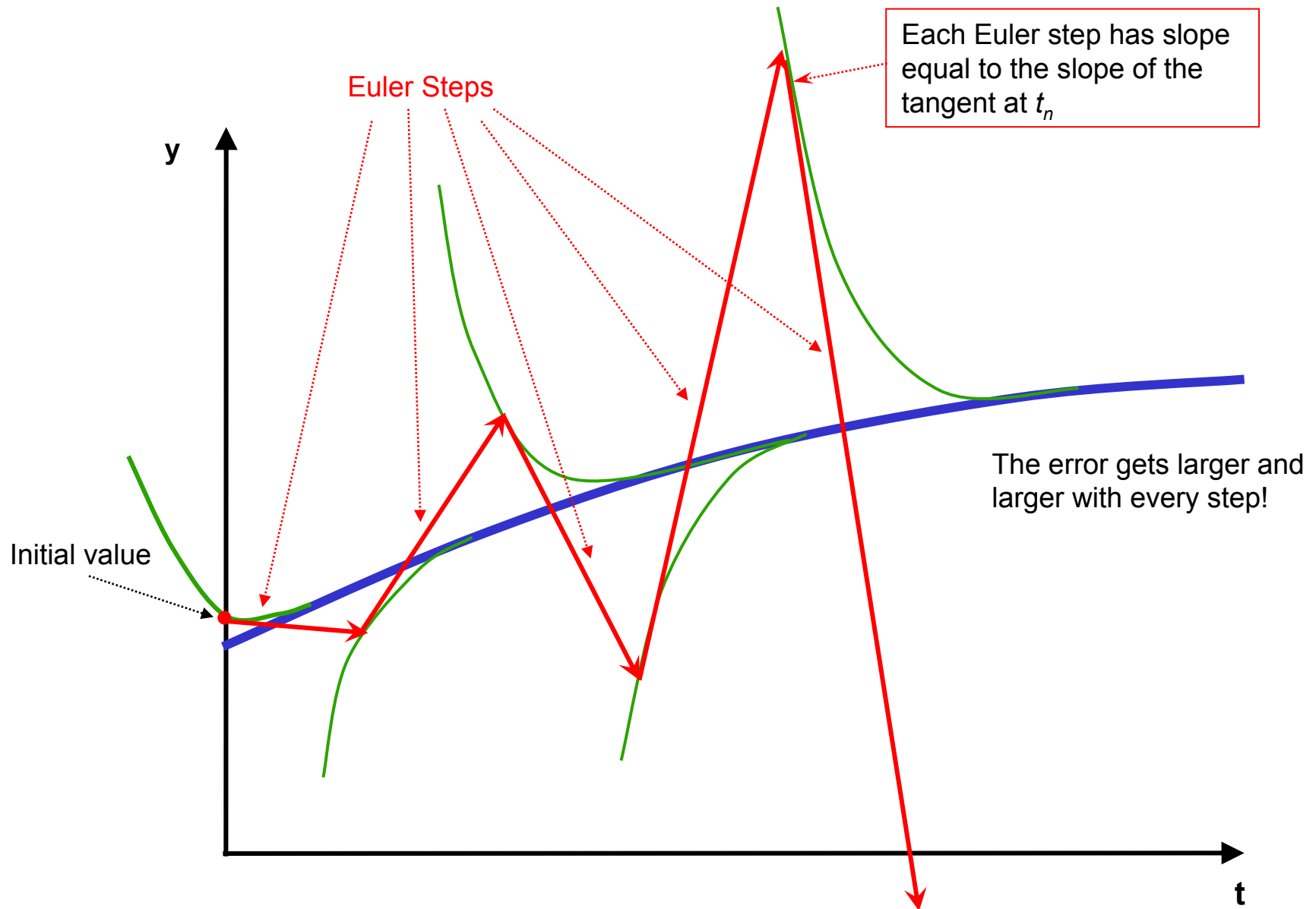
For 15 years stiff equations presented serious difficulties and were hard to solve, both in chemical problems (reaction kinetics) and increasingly in other areas (electrical engineering, mechanical engineering, etc) until around 1968 when a variety of methods began to appear in the literature.

The nature of the problems that leads to *stiffness* is the existence of physical phenomena with very different speeds (time constants) so that, while we may be interested in relative slow aspects of the model, there are features of the model that *could* change very rapidly. Prior to the availability of electronic computers, one could seldom solve problems that were large enough for this to be a problem, but once electronic computers became available and people began to apply them to all sorts of problems, we very quickly ran into *stiff* problems. In fact, most large problems are stiff, as we will see as we look at them in a little detail.

Suppose the family of solutions to an ODE looks like the figure below. This looks to be an ideal problem for integration because almost no matter where we start the final solution finishes up on the blue curve. It shouldn't matter much how big the local error is (within reason) the final answer won't be much different. Unfortunately ...



Consider the Forward Euler applied to this problem starting “close to” but not on the blue curve:



Let us examine a simple equation that illustrates the difficulty:

$$y' = \lambda[y - g(t)] + g'(t), \quad y(0) = g(0)$$

The general solution of this equation is

$$y = g(t) + c \exp(\lambda t)$$

We have chosen initial conditions such that  $c = 0$ . Let us suppose that  $g(t)$  is a smooth, slowly varying function like the blue curve on the previous two slides and that  $\lambda$  is a large, negative number. Then the family of solutions looks like the earlier figure and we expect to see this behavior. Let us look at the error behavior when we use Forward Euler.

Let the global error be  $\varepsilon_n = y_n - g(t_n)$ . Then:

$$\begin{aligned} y_{n+1} &= y_n + h y'_n = y_n + h(\lambda[y_n - g(t_n)] + g'(t_n)) \\ &= g(t_n) + g'(t_n)h + (1 + h\lambda)\varepsilon_n \\ &= g(t_{n+1}) - \frac{h^2}{2} g''(\xi_n) + (1 + h\lambda)\varepsilon_n \end{aligned}$$

or

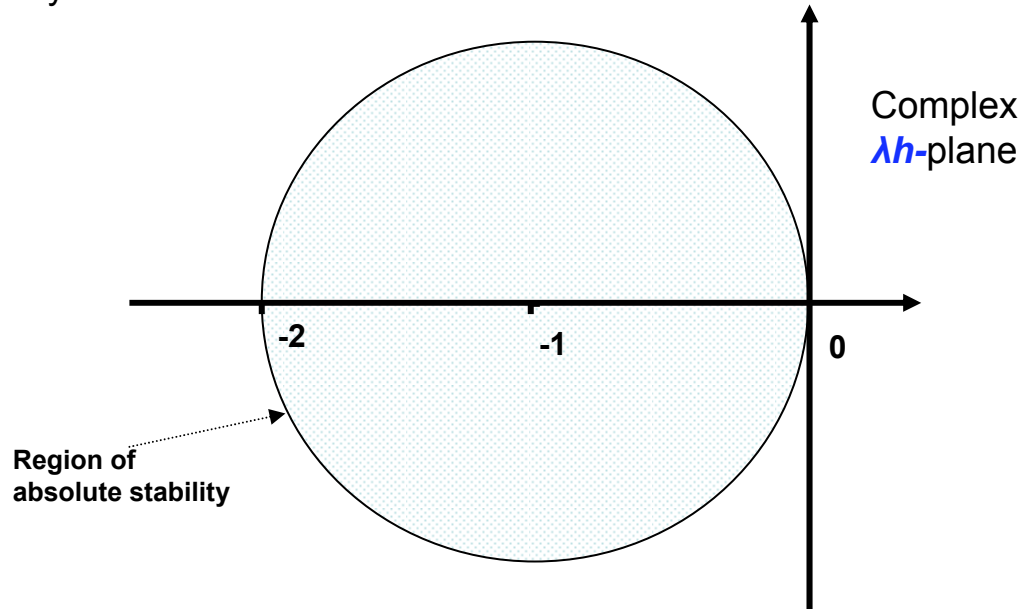
$$\varepsilon_{n+1} = -\frac{h^2}{2} g''(\xi_n) + (1 + h\lambda)\varepsilon_n$$

In this,  $-\frac{h^2}{2} g''(\xi_n)$  is the local error in the step starting from the true solution  $y(t_n) = g(t_n)$ , while  $(1 + h\lambda)\varepsilon_n$  is the error at the  $n$ -th step "amplified" by the method.

Obviously, if  $|(1 + h\lambda)| > 1$  the error grows from step to step.

We say that the method is *absolutely stable* for values of  $h\lambda$  for which  $|(1 + h\lambda)| < 1$

Note that this depends only on the value of  $h\lambda$ , not on the function  $g(t)$  in our last example. Generally we define **absolute stability** for the test equation  $y' = \lambda y$ . It applies to all methods and the **region of absolute stability** is a region in the  $h\lambda$ -plane for which the method does not amplify errors. Note that we consider complex values of  $\lambda$ . This is not because we expect to integrate in the complex domain, but for systems of equations we will see that  $\lambda$  will be the eigenvalue of a matrix, and this can be complex. For the Forward Euler method just discussed, the region of absolute stability is as shown below: it is a disc of radius one centered at -1.



In this example, we want to follow the slowly varying blue curve since that is the solution but there are *fast* components in the system, even though they damp out very quickly. It is evident that if we use Forward Euler we will have to use a very small step size commensurate with the fast components, even though they are not present in the system.

This is the essence of **stiffness**. There are fast time constants in the system, although the corresponding components are no longer present in the solution. However, they force the step size to be small for stability – at least, with the method we have discussed.

The Matlab code below illustrates the blow up in error once  $|(1 + h \lambda)|$  exceeds 1. In this example,  $\lambda$  is -100, and  $h$  runs through the values 0.0199, 0.02, 0.0201, 0.0202, and 0.0203, so that  $(1 + h \lambda)$  is -0.99, -1, -1.01, -1.02, and -1.03. The initial value is deliberately a small amount in error (0.0001).

The errors for the five cases are shown on the next slide. Note that the vertical scale on each plot is different, getting larger the higher on the slide.

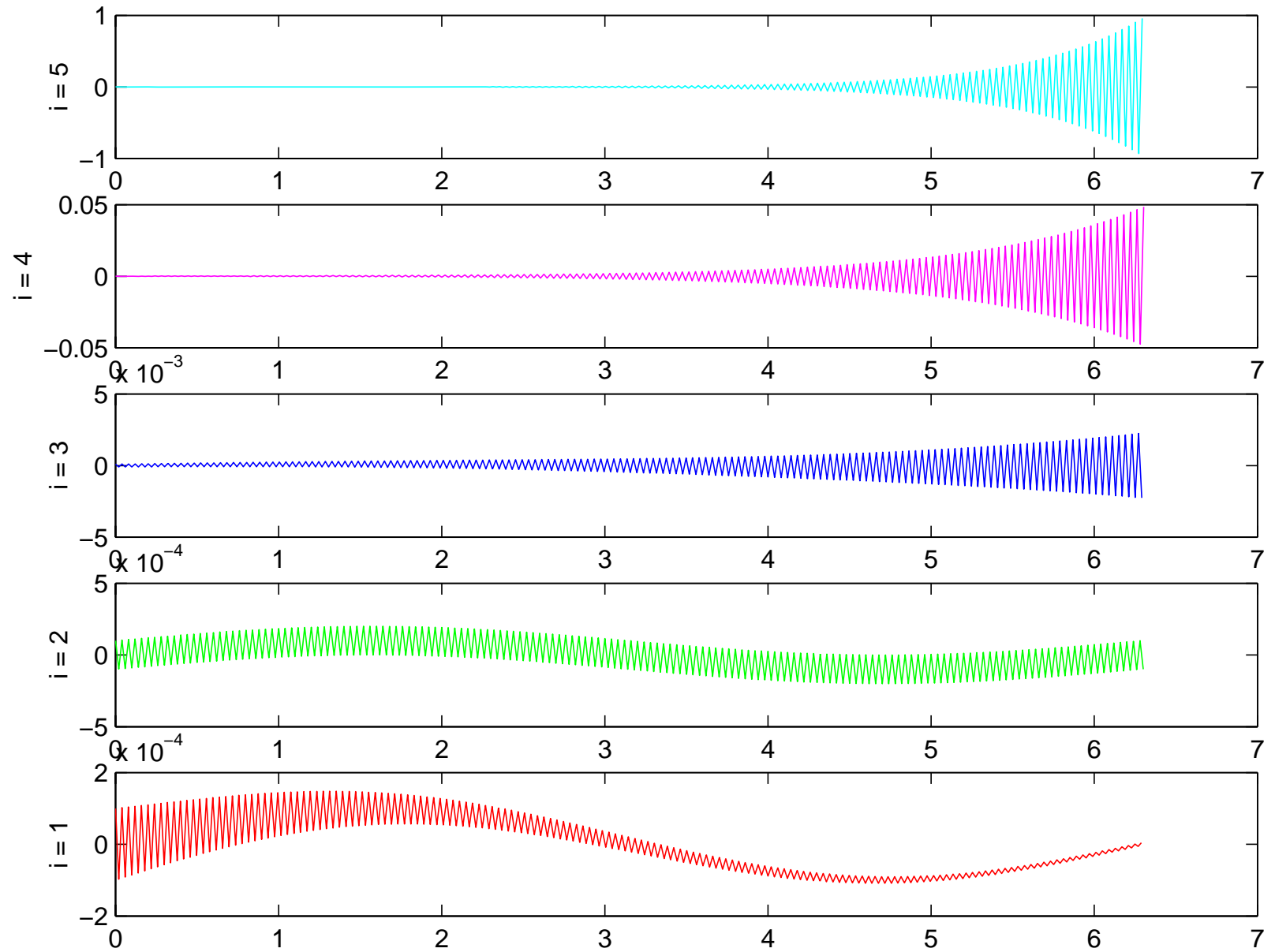
The first case is stable and we see that the initial error is damped, although only slowly since the damping factor  $(1 + h \lambda)$  is -0.99 at each step.

In the second case the error oscillates but doesn't grow since  $(1 + h \lambda)$  is -1.

In the remaining three cases the error grows at ever increasing rates as  $h$  gets bigger.

```
% Example of stiffness effect
% Forward Euler fo y' = -100*(y - sin(t)) + cos(t).
figure(10)
hold off
colorseq = 'rbgmc'      %Colors for plot
for i = 1:5
    h = .02 +(i-2)*.0001; %Vary step size near critical point
    t = 0;                %Initial Value
    y = 0.0001;           %Initial Value - note that it is not sin(0)
    Y = [y];              %Accumulate y for plotting
    T = [t];              %Ditto for t
    while t < 2*pi
        yp = -100*(y-sin(t))+cos(t);
        y = y + h*yp;
        t = t+h;
        Y = [Y;y];
        T = [T; t];
    end
    subplot(5,1,6-i); plot(T,(Y-sin(T)),'-' colorseq(i));
    ylabel(['i = ' num2str(i)])
end
title('Error with different h"s. h = 0.0199: 0.0001 : 0.0203')
```

Error with different h's.  $h = 0.0199 : 0.0001 : 0.0203$



In an earlier slide we only considered a single equation. What if we have a system of equations? Let us consider the linear, constant coefficient case:

$$y' = A[y - g(t)] + g'(t)$$

where  $A$  is a  $p$  by  $p$  matrix. Suppose  $A$  can be transformed to Jordan canonical form by

$$J = SAS^{-1}$$

and let us assume that  $J$  is diagonal (this is not necessary but it simplifies the discussion). By making a change of variables  $z = Sy$  we get the ODE

$$z' = J(z - Sg(t)) + Sg'(t)$$

Since we have assumed that  $J$  is diagonal, this represents  $p$  independent ODEs, each of the form

$$z'_i = \lambda_i(z_i - u_i(t)) + u'_i(t)$$

where  $u_i(t)$  is the  $i$ -th component of the vector  $Sg(t)$  and  $\lambda_i$  is an eigenvalue of  $A$ . Thus we can consider each equation separately. A method will be absolutely stable for this problem if  $h\lambda_i$  is in the region of absolute stability for all  $i$ .

When we have the general non-linear problem

$$y' = f(y) \quad (1)$$

we consider small perturbations. Consider the solution  $y(t) + \varepsilon(t)$ . By subtracting (1) from

$$y'(t) + \varepsilon'(t) = f[y(t) + \varepsilon(t)]$$

we get

$$\varepsilon'(t) = \partial f / \partial y \varepsilon(t)$$

and we see that the eigenvalues of the *Jacobian* matrix  $\partial f / \partial y$  are the values that determine the stability of the method.



We saw that we define *absolute stability* in terms of the test equation  $y' = \lambda y$ . The general solution of this equation is  $A \cdot \exp(\lambda t)$  for an arbitrary constant of integration,  $A$ . If we perturb the solution by a small amount, that perturbation also changes proportionally to  $\exp(\lambda t)$ . How does this function behave as  $t$  increases?

If we split  $\lambda$  into its real and imaginary parts as  $\lambda = \mu + i\nu$  then we see that the solution behaves like

$$A \cdot \exp(\mu t) [\cos(\nu t) + i \sin(\nu t)]$$

Hence, the solution decays whenever the real part of  $\lambda$  is negative. Obviously we might like to find a numerical method that also had this property. Initially this was thought to be a very important property for a method, and it was given a name.

**DEFINITION:** A numerical method is **A-stable** if for the test equation  $y' = \lambda y$  the method has a decreasing solution whenever  $\text{Real}(\lambda) < 0$ .

This region is referred to the “negative half plane” and a method is A-stable if its stability region includes the negative half-plane.

Looking back four slides we see that the stability region of the Forward Euler method includes very little of the negative half plane, and it is not A-stable.

There are A-stable methods, but often their computational cost offsets any advantage they may have. None-the-less, we have to find methods that have better performance than we just saw with Forward Euler. All of the methods we have discussed this far have poor performance for stiff equations, but before we look at other methods, let's look at the stability properties of these methods.

We usually determine the regions of absolute stability by calculating the boundary of that region where the amplification factor is one. We can then determine which parts are stable by a continuity argument – if a point is stable, then any point connected to it by a continuous curve that does not cross the boundary is also stable.

In our plot for the Forward Euler method, we drew the circle where  $|(1 + h\lambda)| = 1$ , and since it is clearly stable when  $h\lambda = -1$  because then  $|(1 + h\lambda)| = 0$ , the interior of this circle is the region of absolute stability.

We can approach the Runge-Kutta methods in the same way. If we ask what happens when we apply the 4<sup>th</sup> order RK method to  $y' = \lambda y$  we will find that we get the recurrence relation

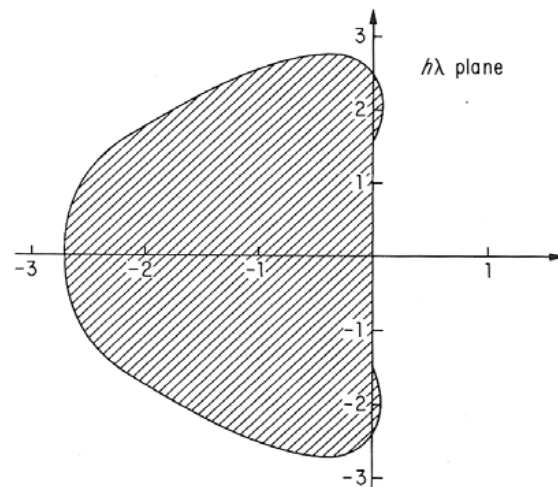
$$y_{n+1} = [1 + h\lambda + (h\lambda)^2/2 + (h\lambda)^3/6 + (h\lambda)^4/24]y_n$$

(It is no accident that the polynomial above is the first five terms of the power series expansion for  $\exp(h\lambda)$ .)

We can find the absolute stability region boundary by finding the values of  $h\lambda$  for which

$$[1 + h\lambda + (h\lambda)^2/2 + (h\lambda)^3/6 + (h\lambda)^4/24]y_n = \exp(i\theta)$$

for  $\theta$  in  $[0, 2\pi]$  (i.e., the amplification has magnitude exactly one). The result is the following. Note that, like the Forward Euler method, it does not go very far into the negative half plane, although it does cover a little more area than the Forward Euler method which became unstable when  $h\lambda < -2$  whereas this is stable until  $h\lambda < -2.8$  approximately.



When we have multi-step methods it is a little easier to plot the regions of absolute stability. If we replace  $y'$  with  $\lambda y$  in the general method below

$$y_{n+1} = \sum_{i=1}^q \alpha_i y_{n+1-i} + h \sum_{i=0}^q \beta_i h y'_{n+1-i}$$

and look for solutions of the form  $y_n = \xi^n$  we get the following equation

$$\xi^{n+1} = \sum_{i=1}^q \alpha_i \xi^{n+1-i} + \sum_{i=0}^q \beta_i h \lambda \xi^{n+1-i}$$

which means we can compute

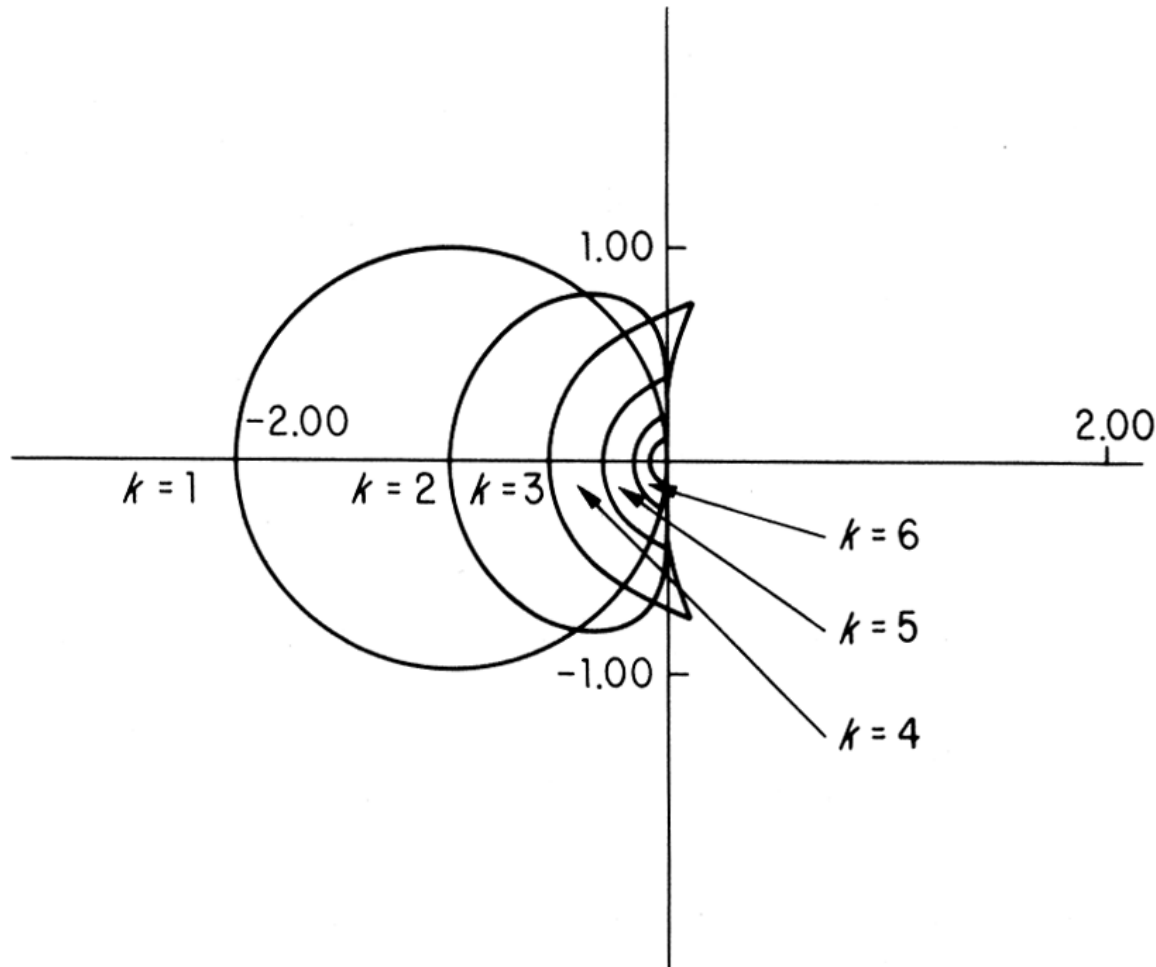
$$h\lambda = [\xi^q - \sum_{i=1}^q \alpha_i \xi^{q-i}] / \sum_{i=0}^q \beta_i \xi^{q-i}$$

to find  $h\lambda$  values on the absolute stability boundary by substituting

$$\xi = \exp(i\theta), \quad \theta \in [0, 2\pi]$$

The next slides show the regions of absolute stability for the Adams Bashforth and Adams Moulton methods discussed in the last lecture

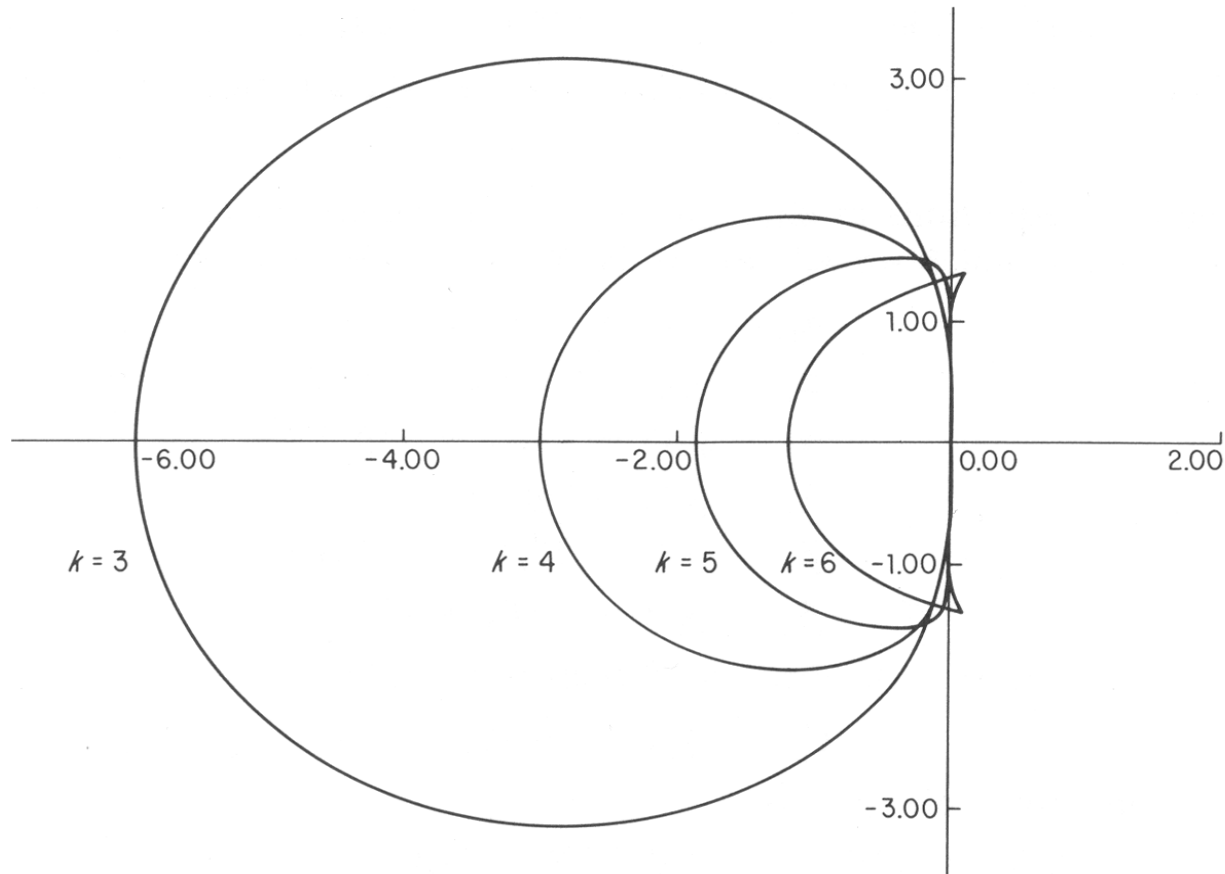
Adams Bashforth Stability regions.  $k = 1$  corresponds to the Forward Euler method – the circular region. Note that they get smaller and smaller as the order,  $k$ , which is equal to the step number gets larger. This means that one would have to use smaller and smaller step sizes as one increased the order if stability was a problem.



Stability region for Adams-Moulton methods of order  $k$  (step number is  $k-1$ ). This stability plot assumes that we have somehow solved the implicit equation exactly, although we haven't yet discussed how.

Note that the regions are much larger than those for the Adams Bashforth method. When we use Adams Moulton in conjunction with Adams Bashforth in a predictor corrector mode, we get a stability region somewhere between the two – not as good as Adams Moulton, and not as bad as Adams Bashforth.

We will be discussing orders 1 and 2 later because they have a special property – they are actually A-stable if you can solve the implicit equation exactly.



Let us go back to the simple equation where we first saw the problem of stiffness and consider what happens as  $\lambda$  gets extremely negative (i.e.,  $\lambda \rightarrow \infty$ ).

$$y' = \lambda[y - g(t)] + g'(t)$$

We see that  $y \rightarrow g(t)$  for all  $t$ . If, instead, we had the ODE

$$y' / \lambda = F(y, t)$$

it is clear that when  $\lambda \rightarrow \infty$  we need to solve the *implicit* equation  $F(y, t) = 0$ . For this reason, if we want to handle arbitrarily large (negative)  $\lambda$  we **have** to use an *implicit* method.

Consider the *Backward Euler Method*:

$$y_{n+1} = y_n + h y'_{n+1}$$

Substituting  $y' = f(y)$  we get

$$y_{n+1} = y_n + h f(y_{n+1})$$

This is an *implicit* equation for  $y_{n+1}$  that has to be solved by some means. In the last lecture we discussed using the Predictor-Corrector functional iteration

$$y_{n+1}^{p+1} = y_n + h f(y_{n+1}^p)$$

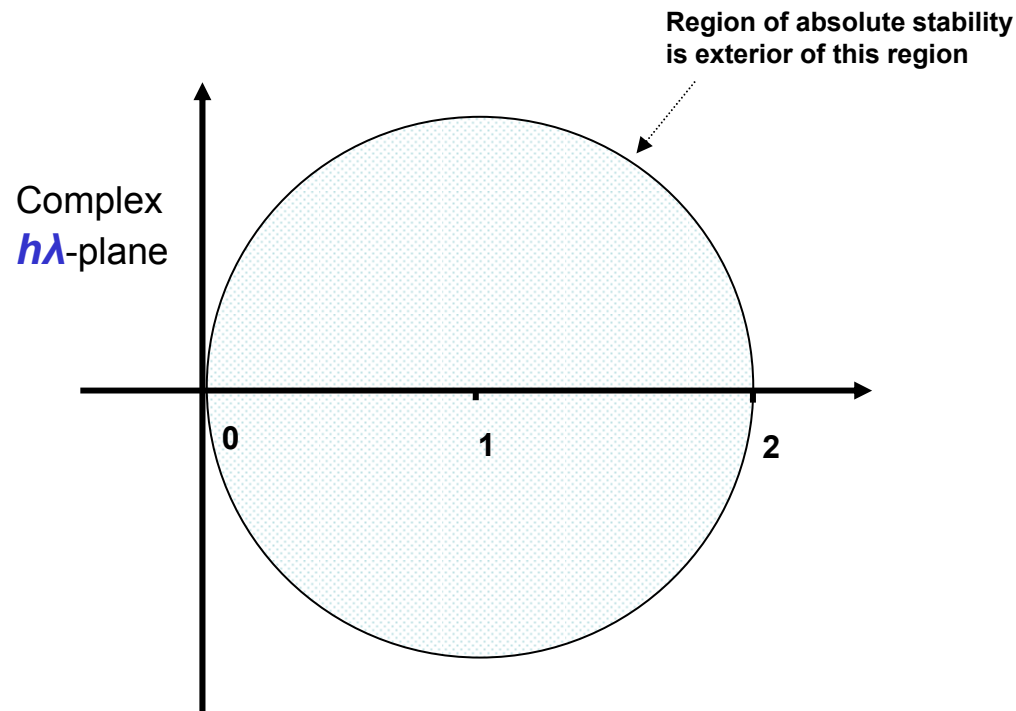
This converges only if  $|h \partial f / \partial y| < 1$ . Note that when  $f(y) = \lambda(y - g(t)) + g'(t)$  then  $\partial f / \partial y = \lambda$ . Hence functional iteration will not work because with stiff equations we have the situation that.

$$|h \partial f / \partial y| = |h \lambda| \gg 1$$

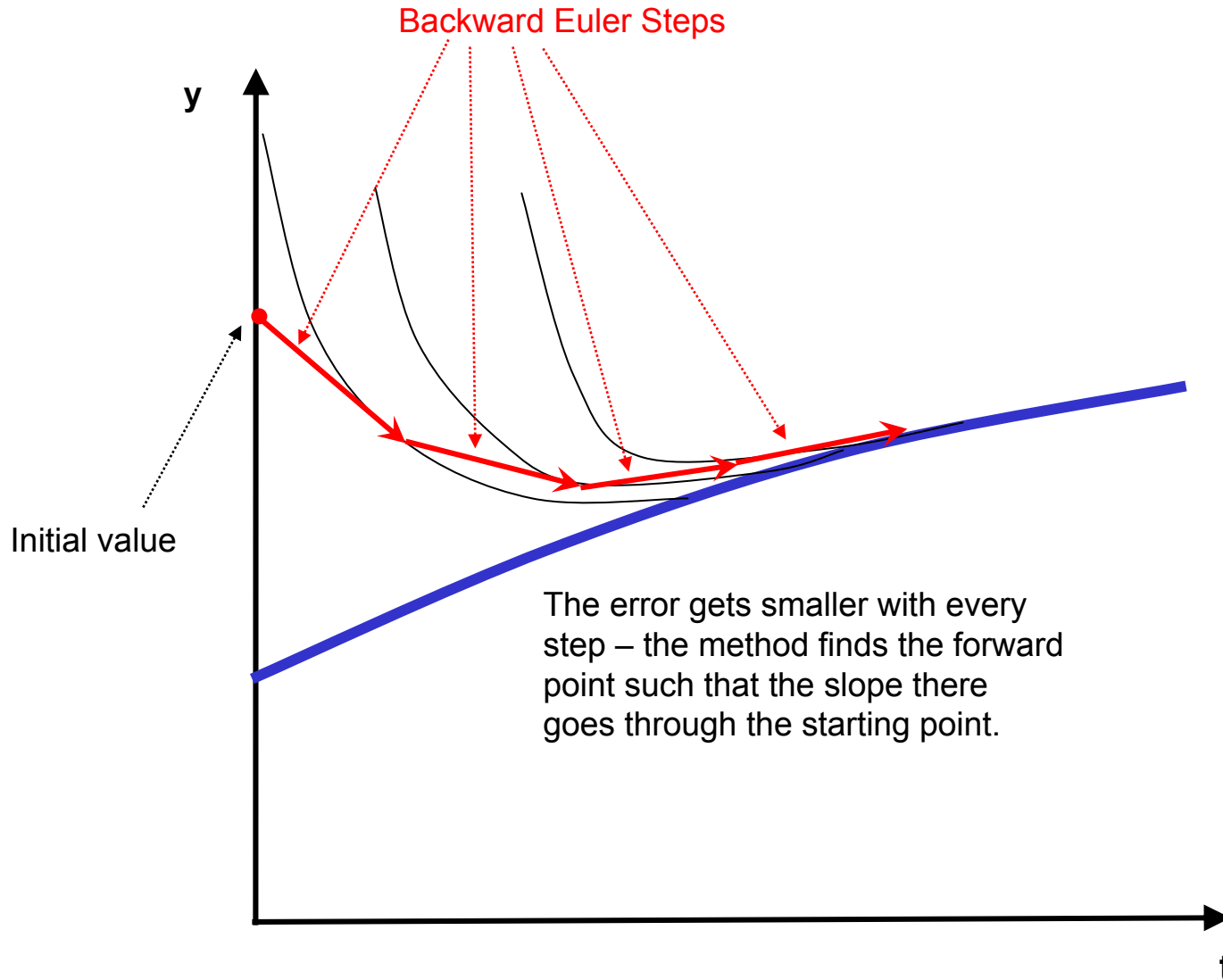
The Backward Euler method is the simplest method that is suitable for stiff equations. Let us assume that somehow we can solve the implicit equation and look at its region of absolute stability. Substituting  $y' = \lambda y$  into  $y_{n+1} = y_n + h y'_{n+1}$  we get

$$y_{n+1} = y_n / (1 - h\lambda)$$

so we see that the amplification factor is  $1 / (1 - h\lambda)$ . This is less than one in magnitude *except* when  $|1 - h\lambda| \leq 1$  which is the interior of the unit circle of radius 1 centered at +1 in the right-half plane as shown below. It is clear that this method is A-Stable because the stability region includes the whole left-half plane.



We can see why the Backward Euler method works for stiff equations graphically, just as we saw earlier why the Forward Euler method is unstable.





## Implicit Equations

We have seen that handling an arbitrarily stiff equation (arbitrarily fast time constants) requires the use of an implicit method. The Backward Euler method illustrates the issues when we have to solve

$$y_{n+1} = y_n + h.f(y_{n+1})$$

for  $y_{n+1}$ .

We have already observed that functional iteration cannot be used because  $h \partial f / \partial y \gg 1$  and so it would not converge. The next obvious choice is Newton's method. this takes the form

$$\left[ I - h \frac{\partial f}{\partial y} \right] [y_{n+1}^{p+1} - y_{n+1}^p] = -[y_{n+1}^p - y_n - hf(y_{n+1}^p)]$$

where  $y_{n+1}^p$  is the  $p$ -th iterate in the Newton iteration. In typical use we get a first guess by using an explicit predictor formula. In most cases, one Newton iteration is sufficient to get accuracy comparable to the accuracy of the implicit corrector formula. Since, for large systems of ODEs the computation of the Jacobian and LU decomposition of the matrix involving it is expensive and because the Jacobian typically changes slowly with time, codes usually do not re-evaluate  $\partial f / \partial y$  unless the Newton iteration does not appear to be converging.

**Note that whereas the computational effort in the integration formula for a system of  $s$  equations is roughly proportional to  $s$ , the computational effort in the Newton iteration can increase like  $s^2$  or  $s^3$ .**

While the solution of the non-linear equations is computationally expensive, it is often much less costly than using an explicit method with very small step sizes for stiff equations.

The Backward Euler Method is A-stable (if we can solve the implicit equation exactly. This method is actually the first-order Adams Moulton method. A  $(q+1)$ -st order Adams Moulton Method has the form

$$y_{n+1} = y_n + h \sum_{i=0}^q \beta_i h y'_{n+1-i}$$

where the  $\beta$  coefficients are chosen to achieve order  $q+1$ . If  $q = 0$ , we see that it is the Backward Euler method. If we choose  $q = 1$  we get the method

$$y_{n+1} = y_n + h(y'_n + y'_{n+1})/2$$

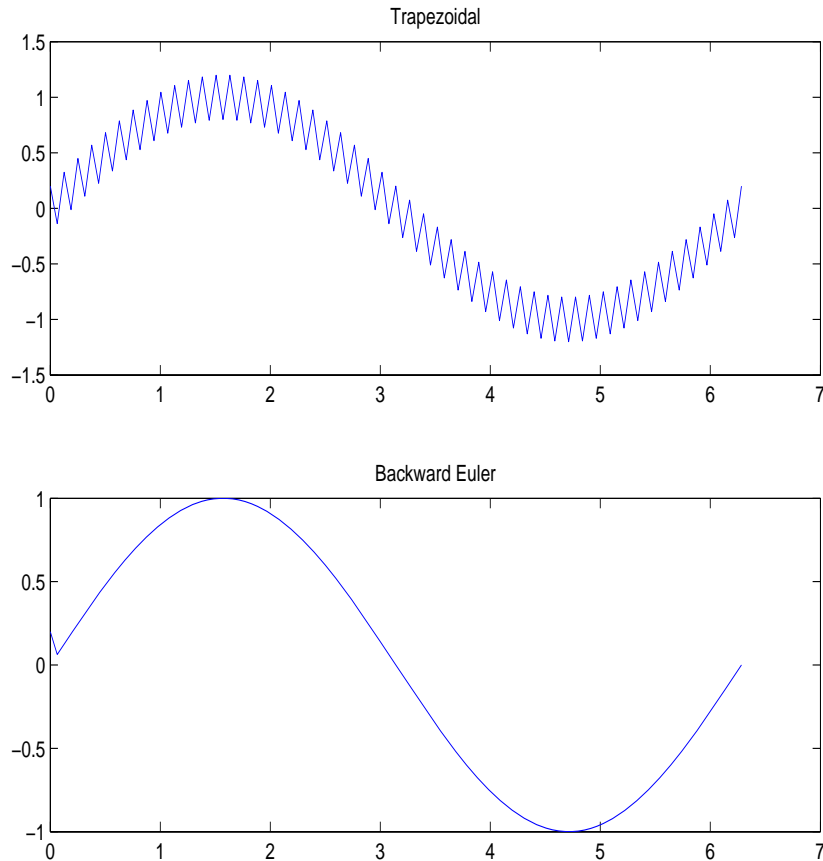
This is known as the Trapezoidal rule. It has order 2 and is also A-Stable. In fact, its region of absolute stability is *exactly* the negative half plane. In that sense, it might be considered an outstanding method for stiff equations – it is stable when they are and it is unstable when they are not.

It also is a good method in another sense, as was shown in another important early paper by Dahlquist (sometimes called Dahlquist's second stability barrier). The theorem states that **the maximum order of a multi-step method that is A-Stable is 2, and that among all order-2 multi-step methods, the one with the smallest local error is the trapezoidal rule.**

So, why look for anything else? For some problems, order 2 is inadequate, and for some problems the trapezoidal rule has an unfortunate property – it does not damp errors in extremely stiff components. We see that by looking at its amplification factor for the usual test equation  $y' = \lambda y$ . It is

$$\frac{1 + h\lambda / 2}{1 - h\lambda / 2}$$

We can see that as  $h\lambda \rightarrow -\infty$  this goes to -1 so that corresponding errors are not damped but oscillate. This is shown on the next slide with a simple code which shows the superiority of the Backward Euler for the problem  $y' = -10^{10}(y - \sin(t)) + \cos(t)$



```
% Ex8 Trapezoidal rule example for  $y' = -1E10*(y - \sin(t)) + \cos(t)$ 
% compared with Backward Euler
lambda = -1E10;
t = 0;
yt = 0.2; %Initial value in error by 0.2
ye = 0.2;
h = 0.01*2*pi; %This is a small enough step for the sin(t) solution.
T = [t];
YE = [ye]; % Backward Euler Solution
YT = [yt]; % Trapezoidal solution
for i = 1:100 % integrate to 2*pi
    t = h*i;
    ye = (ye + h*(-lambda*sin(t) + cos(t)))/(1 - lambda*h);
    yt = (yt*(1+h*lambda/2) + h*(-lambda*(sin(t-h)+sin(t))...
        + cos(t-h)+cos(t))/2)/(1 - lambda*h/2);
    T = [T; t];
    YT = [YT; yt];
    YE = [YE; ye];
end
figure(8)
subplot(2,1,2); plot(T, YE, '-b');
title('Backward Euler')
subplot(2,1,1); plot(T, YT, '-b');
title('Trapezoidal')
print -dpsc Ex8
```

Note that the damping of the Backward Euler method removes the initial error in one step, while the error in the trapezoidal rule oscillates and does not noticeably reduce over the interval.

**NOTE:** The earlier version of this code was in error!

Dalquist's 2<sup>nd</sup> stability barrier says that there are no multi-step methods of order greater than 2 that are A-Stable.

There are two ways around this impasse:

- (1) consider other classes of methods, and
- (2) relax the condition of A-stability.

We will look at both of these options briefly.

(1) The general RK method we discussed earlier is an explicit method and takes the form

$$k_0 = hf(y_n); \quad k_1 = hf(y_n + \beta_{10}k_0); \quad \cdots \quad k_{q-1} = hf(y_n + \sum_{i=0}^{q-2} \beta_{qi}k_i);$$

$$y_{n+1} = y_n + h \sum_{i=0}^{q-1} \gamma_i k_i$$

There is also an *implicit* RK method that has the form

$$k_j = hf(y_n + \sum_{i=0}^{q-1} \beta_{ji}k_i); \quad (1)$$

$$y_{n+1} = y_n + h \sum_{i=0}^{q-1} \gamma_i k_i$$

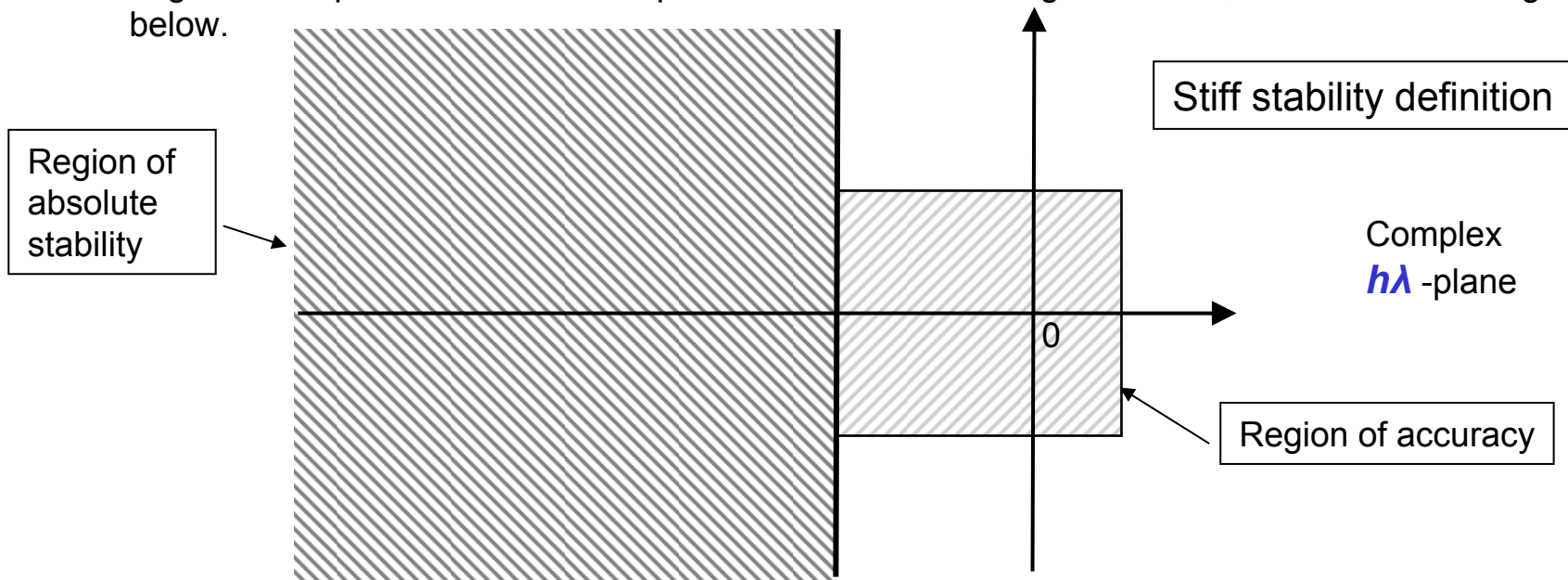
The  $q(q+1)$  coefficients  $\beta_{ji}$  and  $\gamma_j$  of these methods can be chosen to get  $2q$ -th order and these methods can be made A-Stable. (In fact, for  $q = 1$ , this is the trapezoidal method we just discussed.) If the order is  $2q$  then the method does not damp the very fast components – as we just saw for the trapezoidal methods, but by lowering the order we can get more desirable properties. The problem with implicit RK methods is that equation (1) is a set of implicit equations in the  $q$  different  $k_j$ . If the ODE is a system of  $s$  equations, then this is a simultaneous (nonlinear) equation in  $qs$  unknowns which may be expensive to solve since the cost may be proportional to  $(qs)^2$  or  $(qs)^3$ .

Implicit Runge-Kutta (and a few other methods) allow us to get higher-order A-Stable methods, but all at the price of increased matrix arithmetic. An alternative approach to using higher order methods is to relax the requirement of A-Stability. A number less restrictive requirements have been proposed, of which we will mention one – *stiff stability*. Recall that the solution of the test equation  $y' = \lambda y$  when  $\lambda = \mu + i\nu$  is

$$y(t) = A \cdot \exp(\mu t) [\cos(\nu t) + i \sin(\nu t)]$$

In one step  $\nu t$  changes by  $\nu h$ . If the component of the solution corresponding to this is not small, we must limit the size of  $\nu h$  to something less than about 0.1 in order to accurately track the oscillating solution. If  $\mu$  is of a modest size the exponential component will not decay.

Consequently we are not particularly interested in regions of the complex  $h\lambda$ -plane where the real part is not very negative and the complex part is not small since we typically cannot be using a step size that would put us in that region. In a region near the origin we are interested in an accurate solution (since it is not decaying rapidly) while we want absolute stability in that part of the negative half plane where the real part is less than some negative value, as shown in the Figure below.



There are multi-step methods that are stiffly stable for orders greater than 2. The most important of these are what are known as the Backward Differentiation Methods. (Backward Euler is the first order one of this set.)

We introduced Adams methods by approximating the derivative  $y'(t)$  with an interpolating polynomial and integrating it to get  $y$ . In the backward differentiation methods, we approximate  $y(t)$  with an interpolating polynomial and differentiate it to get  $y'(t)$ . We then equate this to the derivative at the most recent point ( $t_{n+1}$ ). Suppose that we have  $q+1$  values of  $y$ , say  $y_{n+1}, y_n, y_{n-1}, \dots, y_{n+1-q}$ . Let the  $q$ -th degree polynomial that passes through these points be

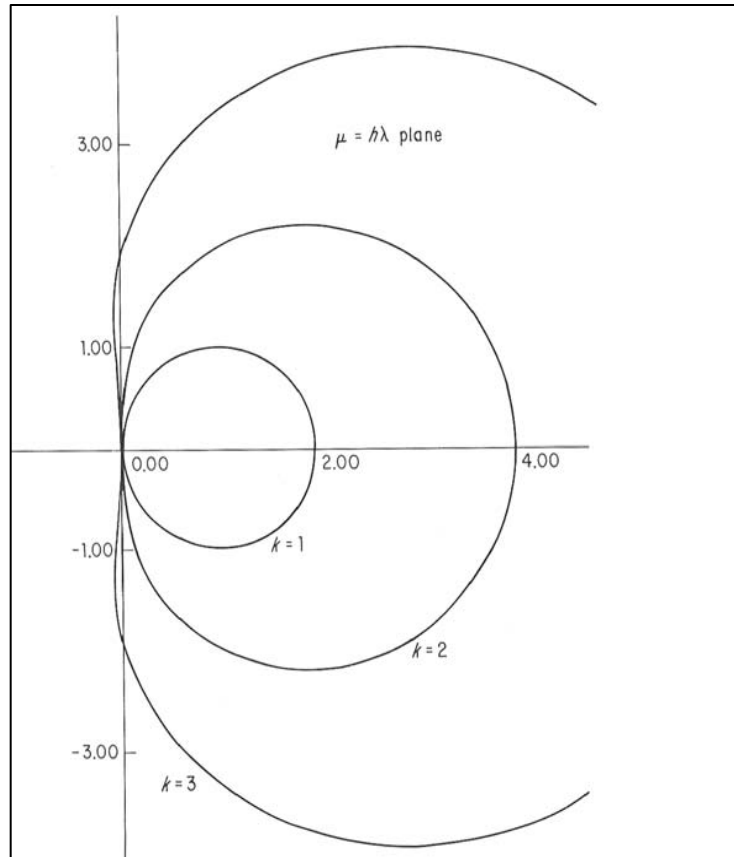
$$p(t) = L_0(t) y_{n+1} + L_1(t) y_n + L_2(t) y_{n-1} + \dots + L_q(t) y_{n+1-q}$$

(This is the Lagrange interpolation formula.) Now let's differentiate this, substitute  $t = t_{n+1}$ , and equate the result to  $f(y_{n+1})$  to get the formula

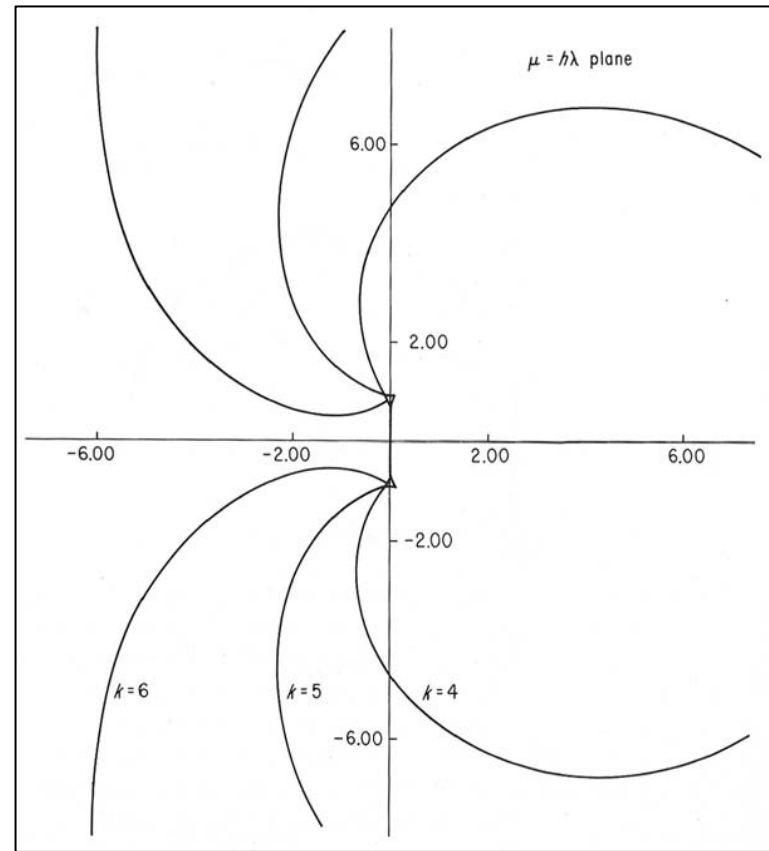
$$\sum_{i=0}^q \alpha_i y_{n+1-i} = hf(y_{n+1})$$

In this formula, we have  $\alpha_i = hL'_i(t_{n+1})$ . This method is of order  $q$ . For  $q < 7$  the methods are stiffly stable. Plots of the boundaries of the regions of absolute stability for BDF methods are shown on the next slide.

## Absolute stability regions for BDF Methods



(a) Order 1 to 3



(b) orders 4 – 6

The regions of stability are the *interiors* of the regions that are primarily in the right-half plane although we have not shown the whole of the boundary (except for the first order method) so as to concentrate on the important region near the origin. Note that as the order increases, the left most point moves further into the left half plane and the boundary begins to “squeeze” around the negative real axis. If we went to order 7, the boundary crosses the negative real axis and the method is no longer stiffly stable, in fact it is not even stable as a method for non-stiff equations.

BDF methods have been used in numerous codes for stiff equations in the same way that Adams methods have been used for non-stiff equations (except, of course, that for stiff equations a Newton-like iteration must be used to solve the corrector equation). Any suitable predictor can be used to get a first guess (and this is really a function of the way the codes are implemented.) The difference between the predictor and the corrector provides an estimate of the error and is used to determine the next step size and order.

Some codes may even make estimates of the largest eigenvalue of the Jacobian during the corrector iteration so as to decide whether to use a stiff or a non-stiff method.

When we used Adams Moulton as a corrector formula we naturally used Adams Bashforth as a predictor. The reason for doing that is that it uses the past derivatives,  $y'_{n-j}$ , *that we have already saved in order to implement the Adams Moulton corrector*. When we use BDF as the corrector for stiff equations, the past values we have saved are the solutions,  $y_{n-j}$  and so it is natural to use these as the basis for the predictor formula. In this case, the predictor is simply an extrapolation through past values of  $y$  so is not really an integration formula, but this does not change the analysis.

In fact, a linear combination of the predictor and corrector can be used as the answer, in which case we get properties somewhere between the two. In the MATLAB program ODE15s, which is a variable order multi-step method for stiff equations, such a combination is the default option (BDF is an alternative option). The combination was chosen to decrease the size of the error coefficient with a slight loss in the region of absolute stability. This is described in [“The MATLAB ODE Suite, by L. F. Shampine and M. W. Reichelt, SIAM Journal on Scientific Computing, 18-1, 1997”](#) which is a good source for some general information on ODE codes.

The point to remember is that countless combinations of methods are possible, and in the end one should pick those that are embedded in high-quality codes.



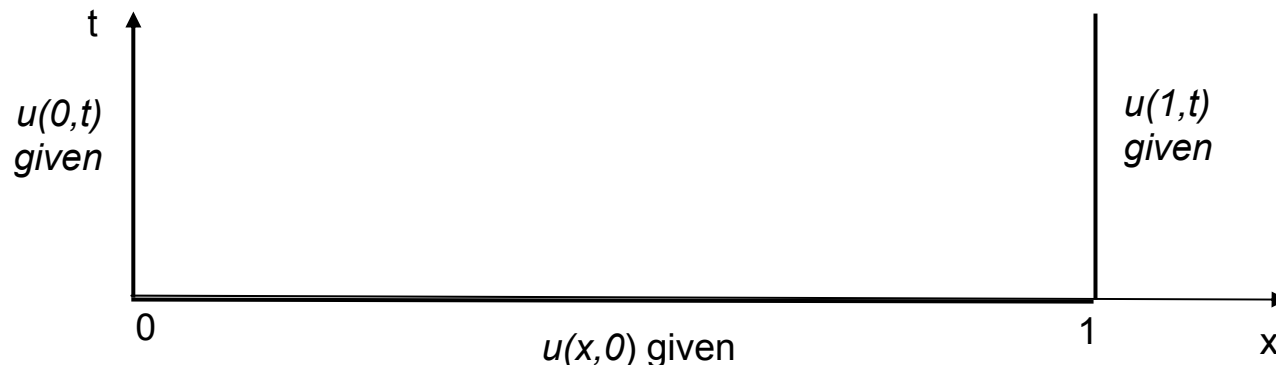
As we noted earlier, small systems of ODEs do not pose many difficulties because the amount of computer time is not excessive. Large problems are the one for which we must look for fast methods, and this is particularly true for stiff equations if we use implicit methods.

Where do large problems come from? There are moderately large problems from, for example, chemical kinetics for systems that involve a very large number species. (These are almost always stiff because some reactions take place very rapidly and others – the ones we usually are interested in following – are slow.

However, these problems are usually small compared to problems that represent *spatially distributed systems*. These are described by Partial Differential Equations (PDEs) but a common way of handling time-dependent PDEs is the *Method of Lines*. Consider a very simple time-dependent PDE – the heat equation. It is:

$$\frac{\partial u}{\partial t} = \mu \frac{\partial^2 u}{\partial x^2}$$

where  $\mu$  is the local heat conductivity. (To keep things simple, we will take  $\mu = 1$  below. Problems like this usually have initial values at, say,  $t = 0$ , and boundary conditions at the ends of the region under consideration. Thus,  $u$  may be specified on the three lines shown below:



In the **method of lines**, we replace the spatial derivatives by finite-dimensional approximations. This can be done by finite differences, finite elements, spectral methods, or any other spatial approximation. Let us consider using a finite difference approximation for the spatial partial derivative. To do this, we *represent* the solution  $u(x,t)$  by its values on a set of lines at various fixed  $x$  values, say  $x_i = i/N$  for  $i = 0, \dots, N$ . Thus we have  **$N+1$**  functions  **$u_i(t) = u(x_i, t)$** . In terms of these we can approximate the spatial partial derivative by

$$\frac{\partial u}{\partial t}(x_i, t) = \frac{u_{i-1}(t) - 2u_i(t) + u_{i+1}(t)}{(\delta x)^2}$$

where  $\delta x$  is the spacing between the “lines” in  $x$  and here is  $1/N$ . Note that  $u_0(t)$  and  $u_N(t)$  are known from the boundary conditions at  $x = 0$  and  $x = 1$ . Hence we have  **$N-1$**  unknowns  $u_i$  that are the solutions of the ODEs

$$\frac{du_i}{dt} = N^2 [u_{i-1} - 2u_i + u_{i+1}]$$

This can be written as an initial value problem for the vector equation in the variable  $\mathbf{u}$ :

$$\frac{d\mathbf{u}}{dt} = A\mathbf{u} + \mathbf{g}(t)$$

where  $\mathbf{g}(t)$  accounts for the boundary conditions and  $A$  is the matrix

$$A = N^2 \begin{bmatrix} -2 & 1 & 0 & & & & \\ 1 & -2 & 1 & \ddots & & & \\ 0 & 1 & -2 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -2 & 1 & 0 \\ & & & \ddots & 1 & -2 & 1 \\ & & & & 0 & 1 & -2 \end{bmatrix}$$

Matrices like the  $N-1$  by  $N-1$  *tri-diagonal* matrix on the right appear frequently when we consider real world simulations problems because it is intimately linked with diffusion, a process that occurs in many models. It has such a simple form that we can write down its eigenvalues and eigenvectors. The easiest way to see what these are is to use some simple trigonometry. We look for an eigenvector,  $\mathbf{x}$ , whose  $i$ -th component is  $\sin(i\pi/N)$  for  $i = 1, \dots, N-1$ .

Since we want  $T\mathbf{x} = \lambda\mathbf{x}$  the  $i$ -th row yields the relation

$$\sin\left(\frac{(i-1)k\pi}{N}\right) - 2\sin\left(\frac{ik\pi}{N}\right) + \sin\left(\frac{(i+1)k\pi}{N}\right) = \lambda \sin\left(\frac{ik\pi}{N}\right)$$

$$T = \begin{bmatrix} -2 & 1 & 0 & & & \\ 1 & -2 & 1 & \ddots & & \\ 0 & 1 & -2 & \ddots & \ddots & \\ & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & \ddots & \ddots & -2 & 1 & 0 \\ & & & \ddots & 1 & -2 & 1 \\ & & & & 0 & 1 & -2 \end{bmatrix}$$

Using the relation  $\sin(A) + \sin(B) = 2\sin\left(\frac{A+B}{2}\right)\cos\left(\frac{A-B}{2}\right)$  we immediately find that

$$\lambda = -2\left[1 - \cos\left(\frac{k\pi}{N}\right)\right]$$

Since this is true for any integer  $k$  we get the  $N-1$  different eigenvalues

with  $k = 1, 2, \dots, N-1$ .

Note, therefore, that the eigenvalues of the matrix  $A$  on the previous slide range from

$$-2N^2\left[1 - \cos\left(\frac{\pi}{N}\right)\right] \text{ to } -2N^2\left[1 - \cos\left(\frac{(N-1)\pi}{N}\right)\right] = -2N^2\left[1 + \cos\left(\frac{\pi}{N}\right)\right]$$

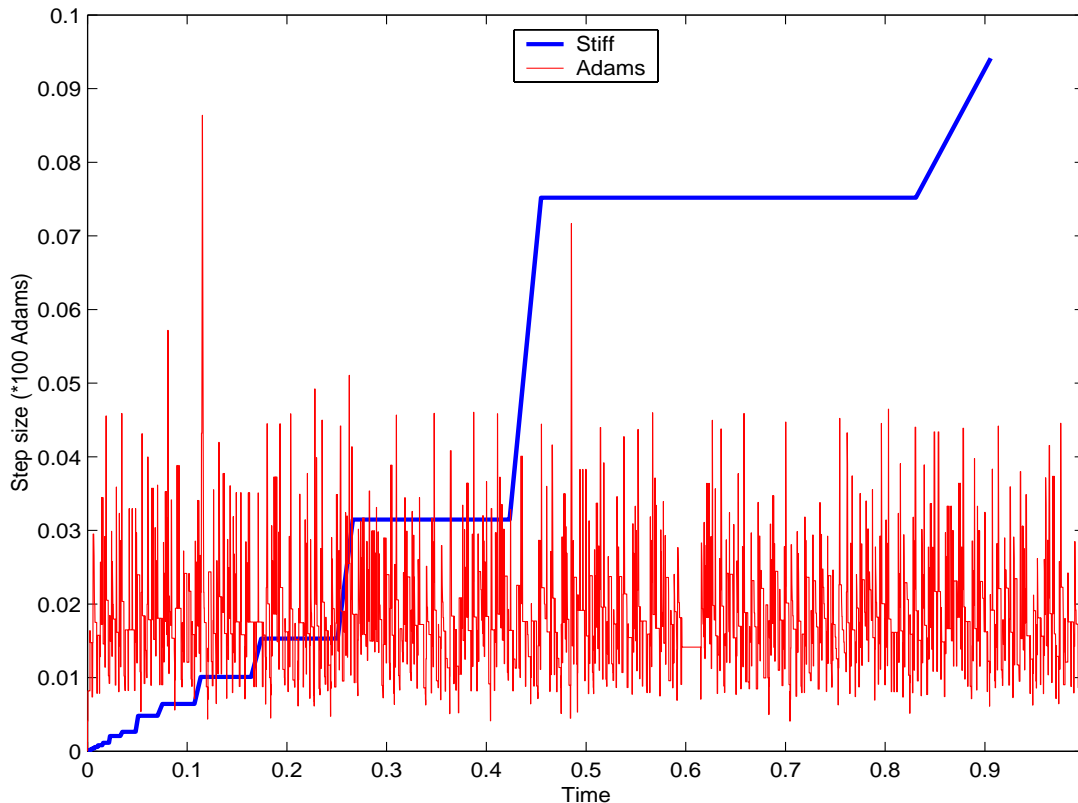
Since  $\cos\left(\frac{\pi}{N}\right) = 1 - \frac{1}{2}\left(\frac{\pi}{N}\right)^2 + \dots$  the eigenvalues range from approximately

$$-\pi^2 \text{ to } -4N^2\left[1 - \frac{1}{4}\left(\frac{\pi}{N}\right)^2\right]$$

Hence, the ODE  $u' = Au + g(t)$  has exponential components with these rates. The eigenvectors of the differential operator on the interval  $[0,1]$  are  $\sin(kx\pi)$  (which is why we guessed the eigenvector for the discrete case). The corresponding solutions to the PDE have the form  $u(x,t) = \exp(\lambda t) \sin(kx\pi)$  where we can easily compute the corresponding eigenvalue as  $-(k\pi)^2$ . Thus we see that the least negative eigenvalue is represented very closely, but as they get more negative the approximation becomes worse.

However, the faster components die out more rapidly so we are less concerned with good approximations to them.

For  $N$  of any size at all, this equation is stiff because it has large negative eigenvalues compared to the typical time scales of interest in the solution. It is interesting to solve this by a stiff and a non-stiff automatic method. In the graph below we show the step sizes used by the codes ode15s (stiff) and ode113 (non stiff – Adams) in Matlab. Note that the step size used by Adams are multiplied by 100 to be visible. The stiff method used 136 steps with no failures, whereas Adams used 6325 steps and rejected an additional 818 others. The stiff method increases the step size slowly, Adams “is noisy.” What we see here is typical of an automatic non-stiff code



when it is given a stiff problem to solve. The code starts the integration with a small step, finds that the solution is very slowly changing and starts to increase the step size. Once the step size gets so large that  $h\lambda$  is no longer in the region of absolute stability, small errors in the solution get amplified by the instability and increase rapidly in size – oscillating as we saw in an earlier slide. This causes the code to think there are now large derivatives and it reduces the step size sharply. Once the step size is small, the method is stable and the errors damp. This process keeps repeating. The final errors in the two solutions are comparable – Adams is a little larger and oscillates. That and the code are shown on the next slide.

### % Ex9 The heat equation using the Method of Lines

% and a non-stiff and a stiff solver.

```
clear
reltol = 1E-4; abstol = 1E-6;
N = 50;
y0 = zeros(N-1,1); %Initial value
options = odeset('RelTol',reltol,'AbsTol',abstol,'Stats','on');
% Stiff solution by modified BDF (NDF)
% Note that the ode solvers in Matlab output the time values
% at all integration points unless one specifies otherwise.
[T,Y] = ode15s(@fun9,[0 1],y0,options);
figure(9)
hold off
Td = T(2:end) - T(1:end-1); %Step sizes are time differences
plot(T(1:end-1),Td,'-b','LineWidth',2)
hold on
% Non stiff solution by Adams
[T,Y] = ode113(@fun9,[0 1],y0,options);
Td = T(2:end) - T(1:end-1);
plot(T(1:end-1),1E2*Td,'-r')
legend('Stiff','Adams',0)
xlabel('Time')
ylabel('Step size (*100 Adams)')
print -dpsc Ex9
```

```
function deriv = fun9(t,y)
```

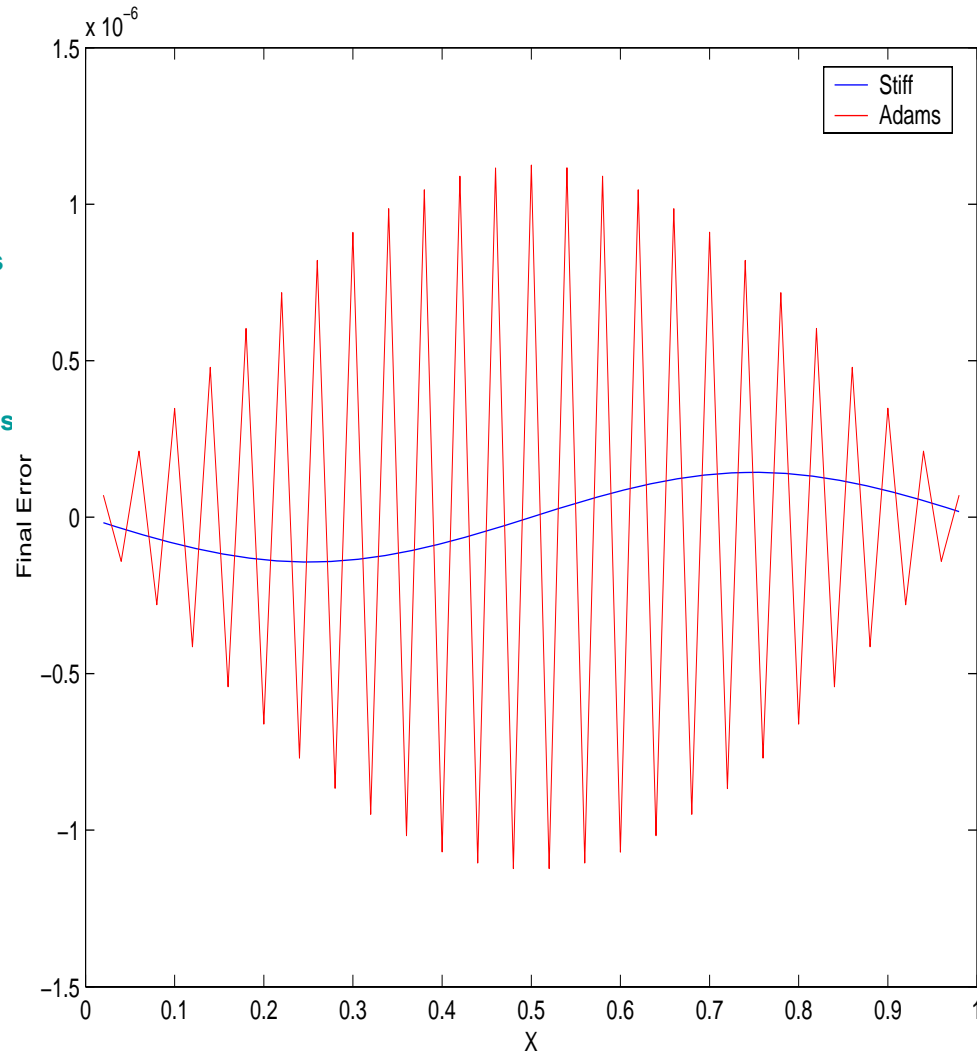
```
%Derivative for the heat equation with
% constant boundary conditions u0 and u1;
```

```
u0 = 0.5; u1 = -0.5;
```

```
N = size(y,1)+1;
```

```
ye = [u0; y; u1]; %y defined on N+1 points
```

```
deriv = N^2*(ye(1:end-2) - 2*ye(2:end-1) + ye(3:end));
end
```



## ERROR CONTROL

There are two types of error control that are commonly used in automatic codes (although the code documentation typically does not tell you which is used. They are called

- **Error per unit step**
- **Error per step**

In **error per unit step**, the estimated local error is controlled to be *approximately* the error tolerance provided by the user scaled by the current step size. The idea behind this is that if errors are neither amplified nor damped by the differential equation in future steps, the global error at the end will be proportional to the length of the integration interval times the tolerance since the global error in this case is just the sum of the local errors so we have:

$$global\_error = \sum_n local\_error_n \leq \sum_n Tol.h_n = Tol.Interval\_Length$$

**Error per step** simply attempts to control each local error to be approximately the tolerance requested. This technique is perhaps best for stiff problems because the differential equation tends to damp early errors strongly. **Error per unit step** is perhaps the most suitable for non-stiff codes, and often occurs by default because of the mechanism used. A typical code estimates the local error using some scheme (such as predictor-corrector difference). It then controls the error to be proportional to the requested tolerance – apparently an error per step approach. However, the code then uses the estimated error to get a hopefully more accurate answer. The error is now one order of  $h$  higher, and involves derivatives of one order higher. If one assumes that the size of the higher-order derivatives are approximately a constant multiple of the derivatives estimated, we now have the actual error proportional to the step size used because of the extra power of  $h$ .

The thought to take away from this discussion is that there can be no guarantee of controlling the error in an automatic code, so that it will be necessary to make multiple runs if it is important to understand the accuracy in your solution.

In the previous example we used the method of lines on a simple one-dimensional PDE. While we may get to fairly large problems when we move to two or three dimensions, we do not get to the really large problems until we have a system of PDEs with many variables in three space dimensions. (Often these are so large that we have to look for symmetries so we can solve them in one or two dimensions as an approximation.

For example, we could be tracking a chemical reaction – such as the burning of fuel – as it moves through space. There will be as many variables as there are species in the reaction, plus variables representing the flow of material. The equations have the form

$$\frac{\partial S_i}{\partial t} = \textit{flow\_terms} + \textit{diffusion\_terms} + \textit{reaction\_terms}$$

with one equation for each species (plus additional equations if we also track pressure, temperature, and velocity). If we use the equivalent of the method of lines we use a large number of cells in space. Within each cell we assume that conditions are uniform and the variables are the various species concentrations. Then we have to follow the chemical reactions in each cell as well as the transport of material from cell to cell. While the chemistry in each cell may only involve a modest number of components, say, 100, if we were solving the problem in three dimensions and used 100 cells in each direction, we would have 100,000,000 variables – a problem size that is becoming challenging!

When problems reach this size, off-the-shelf software becomes problematic because it has to be adapted to the particular problem and to the computer environment in use (problems of this size require large-scale parallelism). Whatever software we use, we have to be aware of the eigenvalues that may be present in the system because they will play a large role in deciding on the best method.

Let us illustrate the issues with a simple problem. Suppose that our “chemistry” involves two species,  $s_1$  and  $s_2$  and let the column vector  $u = [s_1, s_2]^T$ . Suppose that the system is linear so that the chemistry is represented by

$$u' = Ru$$

(This is too simple for any chemistry problem, but will illustrate the issues.) Let us suppose that this is taking place in an *unstirred* liquid where the only other activity is diffusion and that both species diffuse at the same rate, so we have the PDE (which we will restrict to one dimension to simplify the discussion)

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + Ru$$

Let us suppose that we want to model the interval  $[0,1]$  – and this time we will use **cyclic boundary conditions** (the ends meet each other). If we apply the method of lines to this using  $N$  cells we get the matrix system

$$\frac{dv}{dt} = N^2 \begin{bmatrix} R/N^2 - 2I & I & & & I \\ I & R/N^2 - 2I & & & \\ 0 & I & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & I & 0 \\ & & & R/N^2 - 2I & I \\ I & & & I & R/N^2 - 2I \end{bmatrix} v \triangleq Av$$

where  $v = [u_1^T, u_2^T, \dots, u_N^T]^T$  is a column vector of all of the unknowns, and  $I$  is the 2 by 2 identity matrix. In this case we look at eigenvectors of the form

$$w = [x^T \sin(\beta + 2k\pi/N), x^T \sin(\beta + 4k\pi/N), \dots, x^T \sin(\beta + 2Nk\pi/N)]^T$$

where  $x$  is an eigenvector of  $R$  with corresponding eigenvalue  $\nu$ .



If we form  $Aw$  we get for the  $j$ -th pair of rows:

$$xN^2 \sin(\beta + 2jk\pi / N) \left[ \lambda / N^2 - 2 + 2 \cos(2k\pi / N) \right]$$

We get all of the eigenvalues by letting  $k$  range from 1 to  $N$ . Note that we get  $2N$  eigenvalues, namely:  $\lambda - 2N^2[1 - \cos(2k\pi / N)]$  where  $\lambda$  is either of the two eigenvalues of  $R$  and  $k$  ranges from 1 to  $N$ .

Thus we see that the eigenvalues of the compound system are all of the pair-wise sums of the eigenvalues of  $R$  and of the diffusion operator (this time with cyclic boundary conditions which changes them slightly).

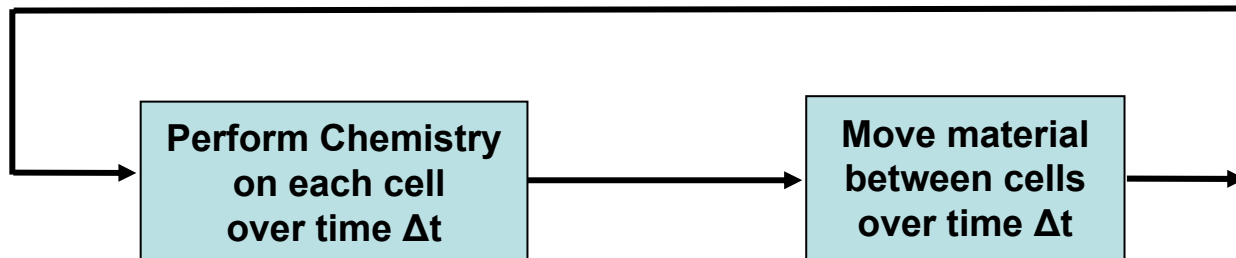
When we have a more complex system (for example, the chemistry is realistic so the Jacobian of the reaction system varies from cell to cell) we no longer can find the eigenvalues explicitly as we did here, but the behavior is similar – there typically will be eigenvalues close to those found by summing the eigenvalues from the two sources.

In situations like this, the stiffness of the system may primarily be determined by the chemistry (since reaction kinetics usually include fast components). However, one needs to understand whether this is true or not. If the eigenvalues of the diffusion operator are a prime cause of stiffness, the best method for solution may be different.

If the eigenvalues due to the chemistry are the dominant ones, the technique of **operator splitting** can often be used to save computation.

Conceptually this amounts to the following view of the computation: we first perform the chemistry in each cell as if it were a stirred reactor with no flow in or out. Having computed the change over one time step, we then perform the computations that handle the flow from cell to cell (whether by diffusion, as in the last example, or by hydrodynamic flow). Since we are handling each cell separately for the chemical kinetics, the size of the implicit system that must be handled in each cell is only the number of species involved. Hence, any matrix arithmetic involved is fairly fast. (In fact, this part can very effectively be handled on separate processors in a parallel processor system.)

It will still be necessary to handle the flow between cells, and, if there is diffusion, it may be necessary to use an implicit method. However, each component of the system can be handled independently so the size of the matrices involved is not as large as the original problem (and, again, they could be handled on separate processors).



We have seen that implicit methods can be very efficient for stiff systems. However, it is important to note that while the computation associated with the evaluations of the derivatives and doing the step-by step integration is roughly proportional to the size of the system,  $s$ , the work involved in solving implicit systems can increase much more rapidly with  $s$ .

For smaller problems, it is almost always best to use *direct* methods for the linear equations that must be solved in each step of the Newton iteration. A *direct* method is usually based on an LU decomposition using Gaussian elimination. If the Jacobian is *dense* (i.e., most of its entries are non-zero) then the LU decomposition takes  $O(s^3)$  operations. The subsequent backsolve takes  $O(s^2)$  operations and is faster.

Recall that the matrix system we are solving at each Newton step is

$$\left[ I - h \frac{\partial f}{\partial y} \right] \Delta y_{n+1}^p = -[y_{n+1}^p - y_n - hf(y_{n+1}^p)]$$

where  $\Delta y$  is the change in  $y$  in one step of the Newton iteration. Computing the Jacobian  $\partial f / \partial y$  also takes additional computer cycles. Codes typically let the user provide a subroutine that evaluates the Jacobian, or will approximate it by numerical differencing. The latter takes  $O(s^2)$  operations also. Since typically  $\partial f / \partial y$  does not change very rapidly, codes usually do not re-evaluate it until they run into convergence problems with the Newton iteration. If the Jacobian  $\partial f / \partial y$  does not change and  $h$  does not change, then the LU decomposition does not change, so it also is re-used from step to step. (Codes also try to avoid frequent step size changes to avoid another LU decomposition. In higher order methods, there is also another constant in front of the  $h$  that depends on past step size ratios and the order of the method which complicates the issue further.)

Regardless of what we do, as the problem dimension gets very large, matrix arithmetic dominates all of operations.

If the system has some structure (as happens when it arises from a PDE, for example) it is worth considering iterative methods. These are often particularly valuable when the Jacobian  $\partial f / \partial y$  is relatively *sparse*, i.e., many of its elements are zero. (This is almost always true when the systems arises from a partial differential equation handled by finite difference or finite element methods or related discretizations in space because non zeros in the Jacobian represent connections between neighboring grid points or elements. It is usually not so if spectral methods are used.)

The reason iterative methods are useful here is that they typically involve only matrix-vector products of the form  $\partial f / \partial y x$  for some small  $x$  which does not require much memory space, whereas an LU decomposition is likely to *fill in* many of the zeros of the matrix and require a lot of space (and time). The product  $\partial f / \partial y x$  can be approximated with the calculation  $f(y+x) - f(y)$  that does not even require an explicit representation of the Jacobian matrix. (Methods based on this are called *Matrix-free methods* and are an important tool in some problems.) Note that this requires that we only want to multiply the Jacobian on the *right* by a vector, not on the left. Methods such as conjugate gradient meet this requirement.

Whatever iterative method is used, convergence is much faster if the matrix can be *pre-conditioned*. This means that we have another matrix,  $M$ , that, in some sense, is an approximation to the inverse of the matrix in our linear system,  $[I - h \partial f / \partial y]$  and for which  $Mz$  can be computed inexpensively. If, for example, we were using the conjugate gradient method, we would like  $M[I - h \partial f / \partial y]$  to have only a few distinct clusters of eigenvalues since the number of iterations is roughly proportional to the number of clusters. For many standard PDEs there are fairly good pre-conditioners in the literature.

In the earlier example of a PDE by the method of lines we used a *parabolic* equation – the heat equation. We saw that it had eigenvalues spread along the negative real axis, leading to a stiff ode problems.

What happens if we start with a hyperbolic equation? Let us consider the one-dimensional wave equation  $u_{tt} = u_{xx}$  as an example. By defining  $v = u_t$  and  $w = u_x$  we get

$$\begin{aligned}\frac{\partial v}{\partial t} &= \frac{\partial w}{\partial x} \\ \frac{\partial w}{\partial t} &= \frac{\partial v}{\partial x}\end{aligned}$$

Again let us use the method of lines on the spatial interval  $[0,1]$  with cyclic boundary conditions and set  $N$  lines at  $x_i = (i - 0.5)/N$  for  $i = 1, 2, \dots, N$ . Clearly,  $\Delta x = 1/N$ . We can now approximate the spatial derivatives to get the system

$$\begin{aligned}\frac{dv_i}{dt} &= \frac{w_{i+1} - w_{i-1}}{2\Delta x} \\ \frac{dw_i}{dt} &= \frac{v_{i+1} - v_{i-1}}{2\Delta x}\end{aligned}$$

where  $v_{N+1} = v_1$ ,  $w_{N+1} = w_1$ ,  $v_0 = v_N$  and  $w_0 = w_N$ . (to handle the boundary conditions). If we now create a column vector  $z = [v_1, w_1, v_2, \dots, v_N, w_N]^T$  containing all the variables we get the system on the next slide

$$\frac{dz}{dt} = \frac{1}{2\Delta x} \begin{bmatrix} 0 & B & 0 & & & 0 & -B \\ -B & 0 & B & \ddots & & & 0 \\ 0 & -B & 0 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & 0 & B & 0 \\ 0 & & & \ddots & -B & 0 & B \\ B & 0 & & & 0 & -B & 0 \end{bmatrix} z \triangleq Az$$

where  $B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

We can find the eigenvalues of the matrix  $A$  by guessing the eigenvectors (they always involve sines and cosines in these sort of problems. In this case we set

$$v_i = \sin(\theta + 2ik\pi/N), \quad w_i = j \cos(\theta + 2ik\pi/N) \quad \text{where } j = \sqrt{-1}$$

for arbitrary  $\theta$  and integer  $k$ . If we look at the  $i$ -th pair of rows of  $Az$  we get

$$\begin{aligned} & \frac{1}{2\Delta x} \begin{bmatrix} -j \cos[\theta + 2(i-1)k\pi/N] + j \cos[\theta + 2(i+1)k\pi/N] \\ -\sin[\theta + 2(i-1)k\pi/N] + \sin[\theta + 2(i+1)k\pi/N] \end{bmatrix} \\ &= \frac{1}{2\Delta x} \begin{bmatrix} -2j \sin(\theta + 2k\pi/N) \sin(2k\pi/N) \\ 2 \cos(\theta + 2ik\pi/N) \sin(2k\pi/N) \end{bmatrix} = \boxed{\frac{-j}{\Delta x} \sin(2k\pi/N)} \begin{bmatrix} \sin(\theta + 2ik\pi/N) \\ j \cos(\theta + 2ik\pi/N) \end{bmatrix} \end{aligned}$$

**Eigenvalue!**

Thus we see that the eigenvalues are all pure imaginary, spread along the imaginary axis out to  $1/\Delta x = N$ .

Some people have called problems with large imaginary eigenvalues “stiff” but I think it is not useful since they present very different difficulties. Imaginary eigenvalues correspond to undamped, high frequency components in the system. If one is certain that the high-frequency components are absent, it could make sense to look for methods that keep them damped, but a lot of time these components may be important and should not be neglected. If the components are present in the system, it is probably necessary to use small steps sizes to resolve them at some point in the calculation. Higher-order methods often do a better job in approximating some of these eigenvalues (look back, for example, to the stability region for the 4-th order RK method on slide L2-10. Note that the boundary follows the imaginary axis quite closely for a while. We will see a reason why this may be true in the discussion that follows.

## CONVERGENCE OF METHODS FOR PDEs.

When we use the method of lines for a PDE we effectively fix the spatial discretization and only consider the set of ODEs that results. We have discussed convergence of methods for ODEs, and so we know that we can select methods that will converge to the solution of the ODEs if we take increasingly small time step sizes. (Of course, we don’t do that in practice.) However, this process does not converge to the solution of the PDE unless we also reduce the spatial discretization. This leads to a whole new set of issues concerning stability (that are addressed in numerous texts on numerical methods for PDEs). Here we will just look at how the issues are different from those in ODEs and what to look out for when using the method of lines.

The standard “prescription” for testing to see if a numerical solution is sufficiently accurate is to reduce the mesh size and compare the results. In the case of partial differential equations handled by the method of lines, this means that one should both check that reducing the time step doesn’t change the solution by more than an acceptable amount, and also that reducing the space mesh also does not reduce the solution by more than an acceptable amount. However, it is probably advisable to integrate both spatial discretizations with two different time steps (or requested error tolerances) for a total of four integrations.

We see why when we look at the error propagation in methods for partial differential equations.

The general form of the error propagation equation for numerical methods for evolutionary differential equations always takes the form

$$\boldsymbol{\varepsilon}_{n+1} = \mathbf{A}_n \boldsymbol{\varepsilon}_n + \mathbf{d}_n \quad (1)$$

where  $\mathbf{A}_n$  is the **error amplification** matrix,  $\mathbf{d}_n$  is the *local error* and  $\boldsymbol{\varepsilon}_n$  is the **global error**.  $\mathbf{A}_n$  determines whether the error grows or decreases from step to step. The matrix involves information from both the method and the equation, and for general methods can be quite complex. In the simple case of Forward Euler for a linear problem, the matrix is the scalar  $(1 + h\lambda)$  – the  $\lambda$  coming from the problem and the linear polynomial from the Forward Euler method.

For a linear, constant coefficient differential equation and with constant step sizes,  $\mathbf{A}_n$  is independent of  $n$  so it is clear that we are concerned with the size of  $\mathbf{A}^n$  as  $n$  increases since we can write the solution of (1) as

$$\boldsymbol{\varepsilon}_n = \sum_{i=0}^{n-1} \mathbf{A}^{n-1-i} \mathbf{d}_i$$

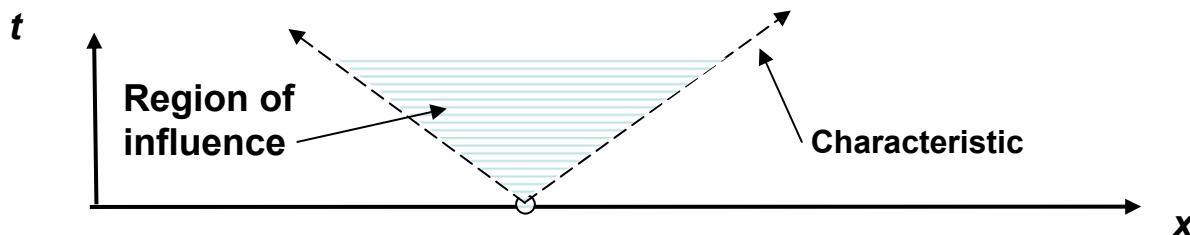


If we want to bound the size of the solution of

$$\varepsilon_n = \sum_{i=0}^{n-1} A^{n-1-i} d_i$$

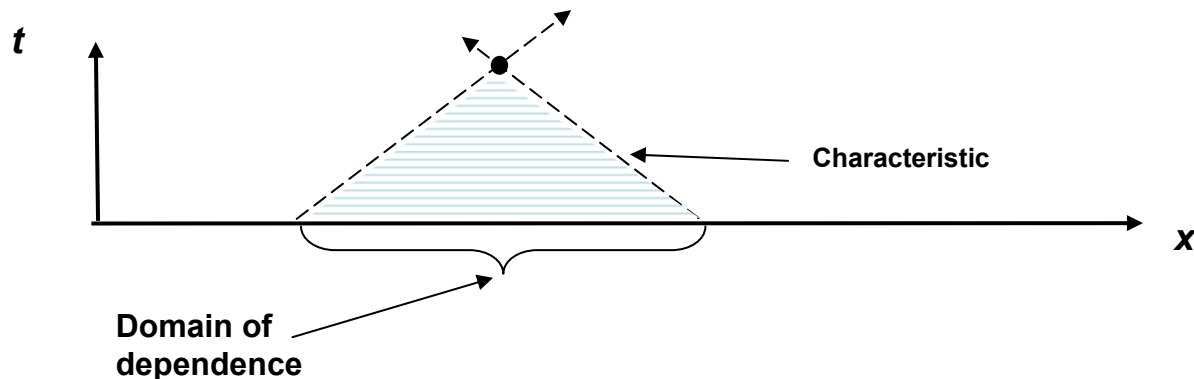
we want to bound  $A^n$ . If  $A$  had eigenvalues all less than one, the problem would be trivial because then we know that  $A$  is power bounded (that is, there is an upper bound on  $A^n$  no matter how large  $n$ ). However,  $A$  necessarily has one eigenvalue that is an approximation to  $\exp(h\lambda)$  so the minor challenge in a convergence proof has to do with showing that  $A^n$  is power bounded over the interval of integration. (That task is usually a little messy but not difficult.)

However, when we have a partial differential equation, the dimension of the matrix  $A$  is dependent on the spatial discretization which is increasing as we decrease the space mesh size. Since with a smaller space mesh size, we have to solve a larger system of ODEs, integrating with the same time step size, or the same error tolerance may not necessarily yield answers of comparable accuracy. We can see why when we look at an important concept in PDEs, particularly in hyperbolic equations (although a related issue arises in parabolic equations). The solution of a hyperbolic equation has *characteristics* which are lines in space along which information is propagated. In the wave equation we discuss earlier, these are the lines  $x+t = c$  and  $x-t = c$  for any values of the constant  $c$ . (For general non-linear PDEs they are much more complex and depend on the solution.) The value of the solution at a given point in space and time,  $(x,t)$ , *influences* only those future points that can be reached by the characteristics as shown below:



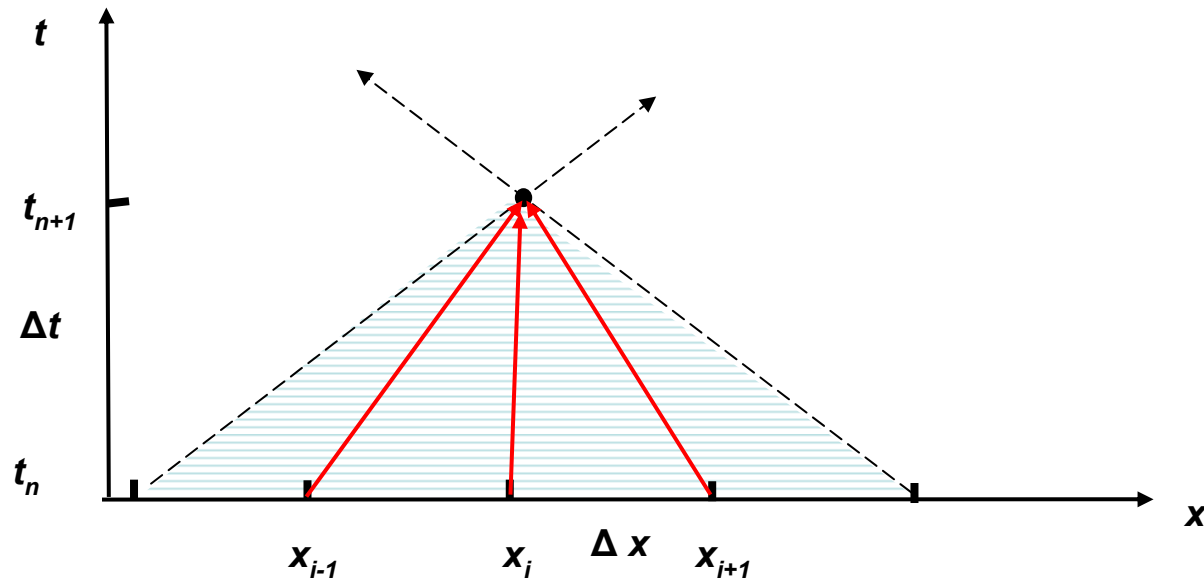
For example, sound propagates at a given speed in a medium, its line of propagation is a characteristic. If you are some distance away from the place lightning strikes, you do not hear the thunder until the sound has had time to travel to you. Until that time you are unaware of the noise.

The region of influence leads to another extremely important concept – what places in the past can influence what is happening to me now. If I am at time  $T$  and position  $x$  I might ask what positions at time  $t < T$  can affect my current state. The answer is the *domain of dependence*. It is also determined by the characteristics as shown below



The solution at the black dot may depend on the values from anywhere within the domain of dependence. (For the simple wave equation it only depends on the values at the end points of the domain of dependence, but this is a special case.) This means that if our numerical method does not make use of information from the whole of the domain of dependence, it cannot possibly converge to the true solution as we reduce the space and time steps.

If, for example, we used a space and time mesh as shown below and used Forward Euler on the simple wave equation discussed earlier, we would only be using information from part of the domain of dependence. (This shows up as instability in the method in this and most cases.) In other words, the time step size has to be limited dependent on the spatial step size. If we use a higher order RK like method that uses multiple function evaluations, it effectively reaches out further, which is why it has better stability along the imaginary axis. If we use implicit methods, the solution of the implicit equation propagates information from all spatial points. However, such methods are often not the best for hyperbolic problems. However if there is fluid flow along with fast reactions, the imaginary eigenvalues due to the flow are usually much smaller than the eigenvalues due to any diffusion and any reaction kinetics, so the flow can usually be handled explicitly.



**For an introductory look at stiff problems, look at the paper:**

**L. F. Shampine and C. W. Gear, A User's View of Solving Stiff Ordinary Differential Equations, SIAM Journal on Numerical Analysis, Vol. 21, No. 1. (Jan., 1979), pp. 1-17, .  
(This is not on my web page.)**

**For extensive mathematical material, reference the book:**

**E. Hairer and G. Wanner, Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Equations, 2nd ed. 1996, Springer-Verlag**