

# Part IV

# Differential Equations

©Copyright, Todd Young and Martin Mohlenkamp, Mathematics Department, Ohio University, 2007

# Lecture 29

## Reduction of Higher Order Equations to Systems

### The motion of a pendulum

Consider the motion of an ideal pendulum that consists of a mass  $m$  attached to an arm of length  $\ell$ . If we ignore friction, then Newton's laws of motion tell us:

$$m\ddot{\theta} = -\frac{mg}{\ell} \sin \theta,$$

where  $\theta$  is the angle of displacement.

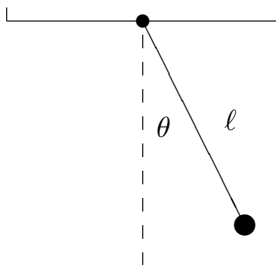


Figure 29.1: A pendulum.

If we also incorporate moving friction and sinusoidal forcing then the equation takes the form:

$$m\ddot{\theta} + \gamma\dot{\theta} + \frac{mg}{\ell} \sin \theta = A \sin \Omega t.$$

Here  $\gamma$  is the coefficient of friction and  $A$  and  $\Omega$  are the amplitude and frequency of the forcing. Usually, this equation would be rewritten by dividing through by  $m$  to produce:

$$\ddot{\theta} + c\dot{\theta} + \omega \sin \theta = a \sin \Omega t, \quad (29.1)$$

where  $c = \gamma/m$ ,  $\omega = g/\ell$  and  $a = A/m$ .

This is a second order ODE because the second derivative with respect to time  $t$  is the highest derivative. It is nonlinear because it has the term  $\sin \theta$  and which is a nonlinear function of the dependent variable  $\theta$ . A solution of the equation would be a function  $\theta(t)$ . To get a specific solution we need side conditions. Because it is second order, 2 conditions are needed, and the usual conditions are initial conditions:

$$\theta(0) = \theta_0 \quad \text{and} \quad \dot{\theta}(0) = v_0. \quad (29.2)$$

## Converting a general higher order equation

All of the standard methods for solving ordinary differential equations are intended for first order equations. For this reason, it is inconvenient to solve higher order equations numerically. However, most higher-order differential equations that occur in applications can be converted to a *system* of first order equations and that is what is usually done in practice.

Suppose that an  $n$ -th order equation can be solved for the  $n$ -th derivative, i.e. it can be written in the form:

$$x^{(n)} = f\left(t, x, \dot{x}, \ddot{x}, \dots, \frac{d^{n-1}x}{dt^{n-1}}\right). \quad (29.3)$$

Then it can be converted to a first-order system by this standard change of variables:

$$\begin{aligned} y_1 &= x \\ y_2 &= \dot{x} \\ &\vdots \\ y_n &= x^{(n-1)} = \frac{d^{n-1}x}{dt^{n-1}}. \end{aligned} \quad (29.4)$$

The resulting first-order system is the following:

$$\begin{aligned} \dot{y}_1 &= \dot{x} = y_2 \\ \dot{y}_2 &= \ddot{x} = y_3 \\ &\vdots \\ \dot{y}_n &= x^{(n)} = f(t, y_1, y_2, \dots, y_n). \end{aligned} \quad (29.5)$$

For the example of the pendulum (29.1) the change of variables has the form

$$\begin{aligned} y_1 &= \theta \\ y_2 &= \dot{\theta}, \end{aligned} \quad (29.6)$$

and the resulting equations are

$$\begin{aligned} \dot{y}_1 &= y_2 \\ \dot{y}_2 &= -cy_2 - \omega \sin(y_1) + a \sin(\Omega t). \end{aligned} \quad (29.7)$$

In vector form this is

$$\dot{\mathbf{y}} = \begin{pmatrix} y_2 \\ -cy_2 - \omega \sin(y_1) + a \sin(\Omega t) \end{pmatrix}.$$

The initial conditions are converted to

$$\mathbf{y}(0) = \begin{pmatrix} y_1(0) \\ y_2(0) \end{pmatrix} = \begin{pmatrix} \theta_0 \\ v_0 \end{pmatrix}. \quad (29.8)$$

As stated above, the main reason we wish to change a higher order equation into a system of equations is that this form is convenient for solving the equation numerically. Most general software for solving ODEs (including MATLAB) requires that the ODE be input in the form of a first-order system. In addition, there is a conceptual reason to make the change. In a system described by a

higher order equation, knowing the position is not enough to know what the system is doing. In the case of a second order equation, such as the pendulum, one must know both the angle and the angular velocity to know what the pendulum is really doing. We call the pair  $(\theta, \dot{\theta})$  the *state* of the system. Generally in applications the vector  $\mathbf{y}$  is the state of the system described by the differential equation.

## Using Matlab to solve a system of ODE's

In MATLAB there are several commands that can be used to solve an initial value problem for a system of differential equations. Each of these correspond to different solving methods. The standard one to use is `ode45`, which uses the algorithm "Runga-Kutta 4 5". We will learn about this algorithm later.

To implement `ode45` for a system, we have to input the vector function  $f$  that defines the system. For the pendulum system (29.7), we will assume that all the constants are 1, except  $c = .1$ , then we can input the right hand side as:

```
> f = inline('[y(2);-.1*y(2)-sin(y(1))+sin(t)]','t','y')
```

The command `ode45` is then used as follows:

```
> [T Y] = ode45(f, [0 20], [1;-1.5]);
```

Here `[0 20]` is the time span you want to consider and `[1;-1.5]` is the initial value of the vector  $\mathbf{y}$ . The output `T` contains times and `Y` contains values of the vector  $\mathbf{y}$  at those times. Try:

```
> size(T)
> T(1:10)
> size(Y)
> Y(1:10,:)
```

Since the first coordinate of the vector is the position (angle), we are mainly interested in its values:

```
> theta = Y(:,1);
> plot(T,theta)
```

In the next two sections we will learn enough about numerical methods for initial value problems to understand roughly how MATLAB produces this approximate solution.

## Exercises

29.1 Transform the given ODE into a first order system:

$$\ddot{x} + \dot{x}^2 - 3\dot{x}^3 + \cos^2 x = e^{-t} \sin(3t).$$

Suppose the initial conditions for the ODE are:  $x(1) = 1$ ,  $\dot{x}(1) = 2$ , and  $\ddot{x}(1) = 0$ . Find a numerical solution of this IVP using `ode45` and plot the first coordinate ( $x$ ). Try time intervals `[1 2]` and `[1 2.1]` and explain what you observe. (Remember to use entry-wise operations, `.*` and `.^`, in the definition of the vector function.)

# Lecture 30

## Euler Methods

### Numerical Solution of an IVP

Suppose we wish to numerically solve the initial value problem

$$\dot{\mathbf{y}} = f(t, \mathbf{y}), \quad \mathbf{y}(a) = \mathbf{y}_0, \quad (30.1)$$

on an interval of time  $[a, b]$ .

By a numerical solution, we must mean an approximation of the solution at a finite number of points, i.e.

$$(t_0, \mathbf{y}_0), (t_1, \mathbf{y}_1), (t_2, \mathbf{y}_2), \dots, (t_n, \mathbf{y}_n),$$

where  $t_0 = a$  and  $t_n = b$ . The first of these points is exactly the initial value. If we take  $n$  steps as above, and the steps are evenly spaced, then the time change in each step is

$$h = \frac{b - a}{n}, \quad (30.2)$$

and the times  $t_i$  are given simply by  $t_i = a + ih$ . This leaves the most important part of finding a numerical solution: determining  $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n$  in a way that is as consistent as possible with (30.1). To do this, first write the differential equation in the indexed notation

$$\dot{\mathbf{y}}_i \approx f(t_i, \mathbf{y}_i), \quad (30.3)$$

and will then replace the derivative  $\dot{\mathbf{y}}$  by a difference. There are many ways we might carry this out and in the next section we study the simplest.

### The Euler Method

The most straight forward approach is to replace  $\dot{\mathbf{y}}_i$  in (30.3) by its forward difference approximation. This gives

$$\frac{\mathbf{y}_{i+1} - \mathbf{y}_i}{h} = f(t_i, \mathbf{y}_i).$$

Rearranging this gives us a way to obtain  $\mathbf{y}_{i+1}$  from  $\mathbf{y}_i$  known as Euler's method:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + hf(t_i, \mathbf{y}_i). \quad (30.4)$$

With this formula, we can start from  $(t_0, \mathbf{y}_0)$  and compute all the subsequent approximations  $(t_i, \mathbf{y}_i)$ . This is very easy to implement, as you can see from the following program (which can be downloaded from the class web site):

```

function [T , Y] = myeuler(f,tspan,y0,n)
% function [T , Y] = myeuler(f,tspan,y0,n)
% Solves dy/dt = f(t,y) with initial condition y(a) = y0
% on the interval [a,b] using n steps of Euler s method.
% Inputs: f -- a function f(t,y) that returns a column vector of the same
%          length as y
%          tspan -- a vector [a,b] with the start and end times
%          y0 -- a column vector of the initial values, y(a) = y0
%          n -- number of steps to use
% Outputs: T -- a n+1 column vector containing the times
%          Y -- a (n+1) by d matrix where d is the length of y
%          Y(t,i) gives the ith component of y at time t
% parse starting and ending points
a = tspan(1);
b = tspan(2);
h = (b-a)/n;      % step size
t = a; y = y0;    % t and y are the current variables
T = a; Y = y0';   % T and Y will record all steps
for i = 1:n
    y = y + h*f(t,y);
    t = a + i*h;
    T = [T; t];
    Y = [Y; y'];
end

```

To use this program we need a function, such as the vector function for the pendulum:

```
> f = inline('[y(2);-.1*y(2)-sin(y(1)) + sin(t)]','t','y')
```

Then type:

```
> [T Y] = myeuler(f,[0 20],[1;-1.5],5);
```

Here [0 20] is the time span you want to consider, [1;-1.5] is the initial value of the vector y and 5 is the number of steps. The output T contains times and Y contains values of the vector as the times. Try:

```
> size(T)
> size(Y)
```

Since the first coordinate of the vector is the angle, we only plot its values:

```
> theta = Y(:,1);
> plot(T,theta)
```

In this plot it is clear that  $n = 5$  is not adequate to represent the function. Type:

```
> hold on
```

then redo the above with 5 replaced by 10. Next try 20, 40, 80, and 200. As you can see the graph becomes increasingly better as  $n$  increases. We can compare these calculations with MATLAB's built-in function with the commands:

```
> [T Y]= ode45(f,[0 20],[1;-1.5]);
> theta = Y(:,1);
> plot(T,theta,'r')
```

## The problem with the Euler method

You can think of the Euler method as finding a linear approximate solution to the initial value problem on each time interval. An obvious shortcoming of the method is that it makes the approximation based on information at the beginning of the time interval only. This problem is illustrated well by the following IVP:

$$\ddot{x} + x = 0, \quad x(0) = 1, \quad \dot{x}(0) = 0. \quad (30.5)$$

You can easily check that the exact solution of this IVP is

$$x(t) = \cos(t).$$

If we make the standard change of variables

$$y_1 = x, \quad y_2 = \dot{x},$$

then we get

$$\dot{y}_1 = y_2, \quad \dot{y}_2 = -y_1.$$

Then the solution should be  $y_1(t) = \cos(t)$ ,  $y_2(t) = \sin(t)$ . If we then plot the solution in the  $(y_1, y_2)$  plane, we should get exactly a unit circle. We can solve this IVP with Euler's method:

```
> g = inline(' [y(2);-y(1)] ', 't', 'y')
> [T Y] = myeuler(g, [0 4*pi], [1;0], 20)
> y1 = Y(:,1);
> y2 = Y(:,2);
> plot(y1,y2)
```

As you can see the approximate solution goes far from the true solution. Even if you increase the number of steps, the Euler solution will eventually drift outward away from the circle because it does not take into account the curvature of the solution.

## The Modified Euler Method

An idea which is similar to the idea behind the trapezoid method would be to consider  $f$  at both the beginning and end of the time step and take the average of the two. Doing this produces the Modified (or Improved) Euler method represented by the following equations:

$$\begin{aligned} \mathbf{k}_1 &= hf(t_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= hf(t_i + h, \mathbf{y}_i + \mathbf{k}_1) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{2}(\mathbf{k}_1 + \mathbf{k}_2). \end{aligned} \quad (30.6)$$

Here  $\mathbf{k}_1$  captures the information at the beginning of the time step (same as Euler), while  $\mathbf{k}_2$  is the information at the end of the time step.

The following program implements the Modified method. It may be downloaded from the class web site.

```

function [T , Y] = mymodeuler(f,tspan,y0,n)
% Solves dy/dt = f(t,y) with initial condition y(a) = y0
% on the interval [a,b] using n steps of the modified Euler's method.
% Inputs: f -- a function f(t,y) that returns a column vector of the same
%          length as y
%          tspan -- a vector [a,b] with the start and end times
%          y0 -- a column vector of the initial values, y(a) = y0
%          n -- number of steps to use
% Outputs: T -- a n+1 column vector containing the times
%          Y -- a (n+1) by d matrix where d is the length of y
%          Y(t,i) gives the ith component of y at time t
% parse starting and ending points
a = tspan(1);
b = tspan(2);
h = (b-a)/n;      % step size
t = a; y = y0;    % t and y are the current variables
T = a; Y = y0';   % T and Y will record all steps
for i = 1:n
    k1 = h*f(t,y);
    k2 = h*f(t+h,y+k1);
    y = y + .5*(k1+k2);
    t = a + i*h;
    T = [T; t];
    Y = [Y; y'];
end

```

We can test this program on the IVP above:

```

> [T Ym] = mymodeuler(g,[0 4*pi],[1;0],20)
> ym1 = Ym(:,1);
> ym2 = Ym(:,2);
> plot(ym1,ym2)

```

## Exercises

30.1 Download the files `myeuler.m` and `mymodeuler.m` from the class web site.

1. Type the following commands:
 

```

> hold on
> f = inline('sin(t)*cos(x)', 't', 'x')
> [T Y]=myeuler(f, [0,12], .1,10);
> plot(T,Y)

```

Position the plot window so that it can always be seen and type:

```

> [T Y]=myeuler(f, [0,12], .1,20);
> plot(T,Y)

```

Continue to increase the last number in the above until the graph stops changing (as far as you can see). Record this number and print the final graph. Type `hold off` and kill the plot window.

2. Follow the same procedure using `mymodeuler.m`.
3. Describe what you observed. In particular compare how fast the two methods converge as  $n$  is increased ( $h$  is decreased).

# Lecture 31

## Higher Order Methods

### The order of a method

For numerical solutions of an initial value problem there are two ways to measure the error. The first is the error of each step. This is called the Local Truncation Error or LTE. The other is the total error for the whole interval  $[a, b]$ . We call this the Global Truncation Error or GTE.

For the Euler method the LTE is of order  $O(h^2)$ , i.e. the error is comparable to  $h^2$ . We can show this directly using Taylor's Theorem:

$$\mathbf{y}(t+h) = \mathbf{y}(t) + h\dot{\mathbf{y}}(t) + \frac{h^2}{2}\ddot{\mathbf{y}}(c)$$

for some  $c$  between  $t$  and  $t+h$ . In this equation we can replace  $\dot{\mathbf{y}}(t)$  by  $f(t, \mathbf{y}(t))$ , which makes the first two terms of the right hand side be exactly the Euler method. The error is then  $\frac{h^2}{2}\ddot{\mathbf{y}}(c)$  or  $O(h^2)$ . It would be slightly more difficult to show that the LTE of the modified Euler method is  $O(h^3)$ , an improvement of one power of  $h$ .

We can roughly get the GTE from the LTE by considering the number of steps times the LTE. For any method, if  $[a, b]$  is the interval and  $h$  is the step size, then  $n = (b-a)/h$  is the number of steps. Thus for any method, the GTE is one power lower in  $h$  than the LTE. Thus the GTE for Euler is  $O(h)$  and for modified Euler it is  $O(h^2)$ .

By the **order** of a method, we mean the power of  $h$  in the GTE. Thus the Euler method is a 1st order method and modified Euler is a 2nd order method.

### Fourth Order Runge-Kutta

The most famous of all IVP methods is the classic Runge-Kutta method of order 4:

$$\begin{aligned} \mathbf{k}_1 &= hf(t_i, \mathbf{y}) \\ \mathbf{k}_2 &= hf(t_i + h/2, \mathbf{y}_i + \mathbf{k}_1/2) \\ \mathbf{k}_3 &= hf(t_i + h/2, \mathbf{y}_i + \mathbf{k}_2/2) \\ \mathbf{k}_4 &= hf(t_i + h, \mathbf{y}_i + \mathbf{k}_3) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \end{aligned} \tag{31.1}$$

Notice that this method uses values of  $f(t, \mathbf{y})$  at 4 different points. In general a method needs  $n$  values of  $f$  to achieve order  $n$ . The constants used in this method and other methods are obtained from Taylor's Theorem. They are precisely the values needed to make all error terms cancel up to  $h^{n+1}f^{(n+1)}(c)/(n+1)!$ .

## Variable Step Size and RK45

If the order of a method is  $n$ , then the GTE is comparable to  $h^n$ , which means it is approximately  $Ch^n$ , where  $C$  is some constant. However, for different differential equations, the values of  $C$  may be very different. Thus it is not easy beforehand to tell how small  $h$  should be to get the error within a given tolerance. For instance, if the true solution oscillates very rapidly, we will obviously need a smaller step size than for a solution that is nearly constant.

How can a program then choose  $h$  small enough to produce the required accuracy? We also do not wish to make  $h$  much smaller than necessary, since that would increase the number of steps. To accomplish this a program tries an  $h$  and tests to see if that  $h$  is small enough. If not it tries again with a smaller  $h$ . If it is too small, it accepts that step, but on the next step it tries a larger  $h$ . This process is called **variable step size**.

Deciding if a single step is accurate enough could be accomplished in several ways, but the most common are called **embedded methods**. The Runge-Kutta 45 method, which is used in `ode45`, is an embedded method. In the RK45, the function  $f$  is evaluated at 5 different points. These are used to make a 5th order estimate  $y_{i+1}$ . At the same time, 4 of the 5 values are used to also get a 4th order estimate. If the 4th order and 5th order estimates are close, then we can conclude that they are accurate. If there is a large discrepancy, then we can conclude that they are not accurate and a smaller  $h$  should be used.

To see variable step size in action, we will define and solve two different ODEs and solve them on the same interval:

```
> f1 = inline('[-y(2);y(1)]','t','y')
> f2 = inline('[-5*y(2);5*y(1)]','t','y')
> [T1 Y1] = ode45(f1,[0 20],[1;0]);
> [T2 Y2] = ode45(f2,[0 20],[1;0]);
> y1 = Y1(:,1);
> y2 = Y2(:,1);
> plot(T1,y1,'bx-')
> hold on
> plot(T2,y2,'ro-')
> size(T1)
> size(T2)
```

## Why order matters

Many people would conclude on first encounter that the advantage of a higher order method would be that you can get a more accurate answer than for a lower order method. In reality, this is not quite how things work. In engineering problems, the accuracy needed is usually a given and it is usually not extremely high. Thus getting more and more accurate solutions is not very useful. So where is the advantage? Consider the following example.

Suppose that you need to solve an IVP with an error of less than  $10^{-4}$ . If you use the Euler method, which has GTE of order  $O(h)$ , then you would need  $h \approx 10^{-4}$ . So you would need about  $n \approx (b - a) \times 10^4$  steps to find the solution.

Suppose you use the second order, modified Euler method. In that case the GTE is  $O(h^2)$ , so you would need to use  $h^2 \approx 10^{-4}$ , or  $h \approx 10^{-2}$ . This would require about  $n \approx (b - a) \times 10^2$  steps.

That is a hundred times fewer steps than you would need to get the same accuracy with the Euler method.

If you use the RK4 method, then  $h^4$  needs to be approximately  $10^{-4}$ , and so  $h \approx 10^{-1}$ . This means you need only about  $n \approx (b - a) \times 10$  steps to solve the problem, i.e. a thousand times fewer steps than for the Euler method.

Thus the real advantage of higher order methods is that they can run a lot faster at the same accuracy. This can be especially important in applications where one is trying to make real-time adjustments based on the calculations. Such is often the case in robots and other applications with dynamic controls.

## Exercises

31.1 There is a Runge-Kutta 2 method, which is also known as the midpoint method. It is summarized by the following equations:

$$\begin{aligned} \mathbf{k}_1 &= hf(t_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= hf(t_i + h/2, \mathbf{y}_i + \mathbf{k}_1/2) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \mathbf{k}_2. \end{aligned} \tag{31.2}$$

Modify the program `mymodeuler` into a program `myRK2` that does the RK2 method. Compare `myRK2` and `mymodeuler` on the following IVP with time span  $[0, 4\pi]$ :

$$\ddot{x} + x = 0, \quad x(0) = 1, \quad \dot{x}(0) = 0.$$

Using `format long` compare the results of the two programs for  $n = 10, 100, 1000$ . In particular, compare the errors of  $\mathbf{x}(4\pi)$ , which should be  $(1, 0)$ .

# Lecture 32

## Multi-step Methods\*

### Exercises

32.1

# Lecture 33

## ODE Boundary Value Problems and Finite Differences

### Steady State Heat and Diffusion

If we consider the movement of heat in a one dimensional object, it is known that the heat,  $u(x, t)$ , at a given location  $x$  and given time  $t$  satisfies the partial differential equation

$$u_t - u_{xx} = g(x, t), \quad (33.1)$$

where  $g(x, t)$  is the effect of any external heat source. The same equation also describes the diffusion of a chemical in a one-dimensional environment. For example the environment might be a canal, and then  $g(x, t)$  would represent how a chemical is introduced.

In many applications, we would only be interested in the steady state of the system, supposing  $g(x, t) = g(x)$  and  $u(x, t) = u(x)$ . In this case

$$u_{xx} = -g(x).$$

This is a linear second-order ordinary differential equation. We could find its solution exactly if  $g(x)$  is not too complicated. If the environment or object we consider has length  $L$ , then typically one would have conditions on each end of the object, such as  $u(0) = 0$ ,  $u(L) = 0$ . Thus instead of an initial value problem, we have a **boundary value problem**.

### Beam With Tension

Consider a simply supported beam with modulus of elasticity  $E$ , moment of inertia  $I$ , a uniform load  $w$ , and end tension  $T$  (see Figure 33.1). If  $y(x)$  denotes the deflection at each point  $x$  in the beam, then  $y(x)$  satisfies the differential equation:

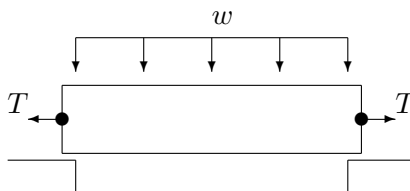
$$\frac{y''}{(1 + (y')^2)^{3/2}} - \frac{T}{EI}y = \frac{wx(L - x)}{2EI}, \quad (33.2)$$

with boundary conditions:  $y(0) = y(L) = 0$ . This equation is nonlinear and there is no hope to solve it exactly. If the deflection is small then  $(y')^2$  is negligible and the equation simplifies to:

$$y'' - \frac{T}{EI}y = \frac{wx(L - x)}{2EI}. \quad (33.3)$$

This is a linear equation and we can find the exact solution. We can rewrite the equation as

$$y'' - \alpha y = \beta x(L - x), \quad (33.4)$$

Figure 33.1: A simply supported beam with a uniform load  $w$  and end tension  $T$ .

where

$$\alpha = \frac{T}{EI} \quad \text{and} \quad \beta = \frac{w}{2EI}, \quad (33.5)$$

and then the exact solution is

$$y(x) = \frac{2\beta}{\alpha^2} \frac{e^{\sqrt{\alpha}L}}{e^{\sqrt{\alpha}L} + 1} e^{-\sqrt{\alpha}x} + \frac{2\beta}{\alpha^2} \frac{1}{e^{\sqrt{\alpha}L} + 1} e^{\sqrt{\alpha}x} + \frac{\beta}{\alpha} x^2 - \frac{\beta L}{\alpha} x + \frac{2\beta}{\alpha^2}. \quad (33.6)$$

### Finite Difference Method – Linear ODE

A finite difference equation is an equation obtained from a differential equation by replacing the variables by their discrete versions and derivatives by difference formulas.

First we will consider equation (33.3). Suppose that the beam is a W12x22 structural steel I-beam. Then  $L = 120$  in.,  $E = 29 \times 10^6$  lb./in.<sup>2</sup> and  $I = 121$  in.<sup>4</sup>. Suppose that the beam is carrying a load of 100,000 lb. so that  $w = 100,000/120 = 10,000$  and a tension of  $T = 10,000$  lb.. For these choices we calculate from (33.5)  $\alpha = 2.850 \times 10^{-6}$  and  $\beta = 1.425 \times 10^{-6}$ . Thus we have the following BVP:

$$y'' = 2.850 \times 10^{-6} y + 1.425 \times 10^{-6} x(120 - x), \quad y(0) = y(120) = 0. \quad (33.7)$$

First subdivide the interval  $[0, 120]$  into four equal subintervals. The nodes of this subdivision are  $x_0 = 0$ ,  $x_1 = 30$ ,  $x_2 = 60$ ,  $\dots$ ,  $x_4 = 120$ . We will then let  $y_0, y_1, \dots, y_4$  denote the deflections at the nodes. From the boundary conditions we have immediately:

$$y_0 = y_4 = 0.$$

To determine the deflections at the interior points we will rely on the differential equation. Recall the central difference formula

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}.$$

In this case we have  $h = (b - a)/n = (120 - 0)/4 = 30$ . Replacing all the variables in the equation (33.4) by their discrete versions we get

$$y_{i+1} - 2y_i + y_{i-1} = h^2 \alpha y_i + h^2 \beta x_i (L - x_i).$$

Substituting in for  $\alpha$ ,  $\beta$  and  $h$  we obtain:

$$\begin{aligned} y_{i+1} - 2y_i + y_{i-1} &= 900 \times 2.850 \times 10^{-6} y_i + 900 \times 1.425 \times 10^{-6} x_i (120 - x_i) \\ &= 2.565 \times 10^{-3} y_i + 1.282 \times 10^{-3} x_i (120 - x_i). \end{aligned}$$

This equation makes sense for  $i = 1, 2, 3$ . At  $x_1 = 30$ , the equation becomes:

$$\begin{aligned} y_2 - 2y_1 + y_0 &= 2.565 \times 10^{-3}y_1 + 1.282 \times 10^{-3} \times 30(90) \\ \Leftrightarrow y_2 - 2.002565y_1 &= 3.463. \end{aligned} \quad (33.8)$$

Note that this equation is linear in the unknowns  $y_1$  and  $y_2$ . At  $x_2 = 60$  we have:

$$\begin{aligned} y_3 - 2y_2 + y_1 &= .002565y_2 + 1.282 \times 10^{-3} \times 60^2 \\ \Leftrightarrow y_3 - 2.002565y_2 + y_1 &= 4.615. \end{aligned} \quad (33.9)$$

At  $x_3 = 90$  we have (since  $y_4 = 0$ )

$$-2.002565y_3 + y_2 = 3.463. \quad (33.10)$$

Thus  $(y_1, y_2, y_3)$  is the solution of the linear system:

$$\left( \begin{array}{ccc|c} -2.002565 & 1 & 0 & 3.463 \\ 1 & -2.002565 & 1 & 4.615 \\ 0 & 1 & -2.002565 & 3.463 \end{array} \right).$$

We can easily find the solution of this system in MATLAB:

```
> A = [ -2.002565 1 0 ; 1 -2.002565 1 ; 0 1 -2.002565]
> b = [ 3.463 4.615 3.463 ]'
> y = A\b
```

To graph the solution, we need define the  $x$  values and add on the values at the endpoints:

```
> x = 0:30:120
> y = [0 ; y ; 0]
> plot(x,y,'d')
```

Adding a spline will result in an excellent graph.

The exact solution of this BVP is given in (33.6). That equation, with the parameter values for the W12x22 I-beam as in the example, is in the program `myexactbeam.m` on the web site. We can plot the true solution on the same graph:

```
> hold on
> myexactbeam
```

Thus our numerical solution is extremely good considering how few subintervals we used and how very large the deflection is.

An amusing exercise is to set  $T = 0$  in the program `myexactbeam.m`; the program fails because the exact solution is no longer valid. Also try  $T = .1$  for which you will observe loss of precision. On the other hand the finite difference method works when we set  $T = 0$ .

## Exercises

33.1 Derive the finite difference equations for the BVP (33.7) on the same domain ( $[0, 120]$ ), but with eight subintervals and solve (using MATLAB) as in the example. Plot your result, together with the exact solution (33.6) from the program `myexactbeam.m`.

33.2 Derive the finite difference equation for the BVP:

$$y'' + y' - y = x \quad \text{with} \quad y(0) = y(1) = 0$$

with 4 subintervals. Solve them and plot the solution using MATLAB.

# Lecture 34

## Finite Difference Method – Nonlinear ODE

### Heat conduction with radiation

If we again consider the heat in a bar of length  $L$ , but this time consider the effect of radiation as well as conduction, then the steady state equation has the form

$$u_{xx} - d(u^4 - u_b^4) = -g(x), \quad (34.1)$$

where  $u_b$  is the temperature of the background,  $d$  incorporates a coefficient of radiation and  $g(x)$  is the heat source.

If we again replace the continuous problem by its discrete approximation then we get

$$\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - d(u_i^4 - u_b^4) = -g_i = -g(x_i). \quad (34.2)$$

This equation is nonlinear in the unknowns, thus we no longer have a system of linear equations to solve, but a system of nonlinear equations. One way to solve these equations would be by the multivariable Newton method. Instead, we introduce another iterative method, known as the *relaxation* method.

### Relaxation Method for Nonlinear Finite Differences

We can rewrite equation (34.2) as

$$u_{i+1} - 2u_i + u_{i-1} = h^2 d(u_i^4 - u_b^4) - h^2 g_i.$$

From this we can solve for  $u_i$  in terms of the other quantities:

$$2u_i = u_{i+1} + u_{i-1} - h^2 d(u_i^4 - u_b^4) + h^2 g_i.$$

Next we add  $u_i$  to both sides of the equation to obtain

$$3u_i = u_{i+1} + u_i + u_{i-1} - h^2 d(u_i^4 - u_b^4) + h^2 g_i,$$

and then divide by 3 to get

$$u_i = \frac{1}{3}(u_{i+1} + u_i + u_{i-1}) - \frac{h^2}{3}(d(u_i^4 - u_b^4) + g_i).$$

Now for the main idea. We will begin with an initial guess for the value of  $u_i$  for each  $i$ , which we can represent as a vector  $\mathbf{u}^0$ . Then we will use the above equation to get better estimates,  $\mathbf{u}^1$ ,  $\mathbf{u}^2$ ,  $\dots$ , and hope that they converge to the correct answer.

If we let

$$\mathbf{u}^j = (u_0^j, u_1^j, u_2^j, \dots, u_{n-1}^j, u_n^j)$$

denote the  $j$ th approximation, then we can obtain that  $j + 1$ st from the formula

$$u_i^{j+1} = \frac{1}{3} (u_{i+1}^j + u_i^j + u_{i-1}^j) - \frac{h^2}{3} (d((u_i^j)^4 - u_b^4) + g_i).$$

Notice that  $g_i$  and  $u_b$  do not change. In the resulting equation, we have  $u_i$  at each successive step depending on its previous value and the equation itself.

## Fixed Boundary Values

If we have fixed boundary conditions. i.e.  $u(0) = a$ ,  $u(L) = b$ , then we will make  $u_0^j = a$  and  $u_n^j = b$  for all  $j$ .

## Implementing the Relaxation Method

In the following program we solve the finite difference equations (34.2) with the boundary conditions  $u(0) = 0$  and  $u'(L) = 0$ . We let  $L = 4$ ,  $n = 4$ ,  $d = 1$ , and  $g(x) = \sin(\pi x/4)$ . Notice that the grid includes the point  $L + h$ . Notice that the vector  $\mathbf{u}$  always contains the current estimate of the values of  $\mathbf{u}$ .

```
% mynonlinheat (lacks comments)
L = 4;    n = 4;
h = L/n;  hh = h^2/3;
u0 = 0;   ub = .5;  ub4 = ub^4;
x = 0:h:L+h;  g = sin(pi*x/4);  u = zeros(1,n+2);
steps = 4;
u(1)=u0;
for j = 1:steps
    u(2:n+1) = (u(3:n+2)+u(2:n+1)+u(1:n))/3 + hh*(-u(2:n+1).^4+ub4+g(2:n+1));
    u(n+2) = u(n);
end
plot(x,u)
```

If you run this program the result will not seem reasonable.

We can plot the initial guess by adding the command `plot(x,u)` right before the `for` loop. We can also plot successive iterations by moving the last `plot(x,u)` before the `end`. Now we can experiment and see if the iteration is converging. Try various values of `steps` and `n` to produce a good plot. You will notice that this method converges quite slowly. In particular, as we increase  $n$ , we need to increase `steps` like  $n^2$ , i.e. if  $n$  is large then `steps` needs to be *really* large.

**Exercises**

- 34.1 Modify the script program `mynonlinheat` to plot the initial guess and all intermediate approximations and add complete comments to the program. Also modify it so that the boundary conditions are

$$u(0) = 0 \quad \text{and} \quad u(L) = 10.$$

Print the program and a plot using large enough `n` and `steps` to see convergence.

# Lecture 35

## Boundary Conditions

### Boundary Conditions

In the previous lab we used fixed boundary conditions. i.e.  $u(0) = a$ ,  $u(L) = b$ , then we will make  $u_0^j = a$  and  $u_n^j = b$  for all  $j$ .

If we have an insulated end, say at the right end, then we want to replace  $u'(L) = 0$ , by a discrete version. If we use the most accurate version, the central difference, then we should have

$$u'(L) = u'(x_n) = \frac{u_{n+1} - u_{n-1}}{2h} = 0.$$

or more simply

$$u_{n+1} = u_{n-1}.$$

However,  $u_{n+1}$  would represent  $u(x_{n+1})$  and  $x_{n+1}$  would be  $L + h$ , which is outside the bar. This however is not an obstacle in a program. We can simply extend the grid to one more node,  $x_{n+1}$ , and let  $u_{n+1}$  always equal  $u_{n-1}$ . This idea is carried out in the program of the next section.

Another practical way to implement an insulated boundary is to let the grid points straddle the boundary. For example if  $u'(0) = 0$ , then you could let the first two grid points be at  $-h/2$  and  $h/2$ . Then you can let

$$u_0 = u_1.$$

This will again force the central difference at  $x = 0$  to be 0.

A way to think of an insulated boundary that makes sense of the point  $L + h$  is to think of two bars joined end to end, where you do the mirror image of the first bar to the second bar. If you do this, then no heat will flow across the joint, which is exactly the same effect as insulating.

### An example

The steady state temperature  $u(r)$  (given in polar coordinates) of a disk subjected to a radially symmetric heat load  $g(r)$  and cooled by conduction to the rim of the disk and radiation to its environment is determined by the boundary value problem:

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r} \frac{\partial u}{\partial r} = d(u^4 - u_b^4) - g(r) \quad \text{with} \quad u(R) = u_R \quad \text{and} \quad u'(0) = 0.$$

Here  $u_b$  is the background temperature and  $u_R$  is the temperature at the rim of the disk.

Suppose  $R = 5$ ,  $d = .1$ ,  $u_R = u_b = 10$ ,  $g(r) = (r - 5)^2$  and derive the finite difference equations for this BVP. Then solve for  $u_i$  as we did above to get a relaxation update formula.

The class web site has a program `myheatdisk.m` that implements these equations. Notice that the equations have a singularity at  $r = 0$ . How does the program avoid this problem? How does the program implement  $u'(0) = 0$ ? Run the program.

## Exercises

- 35.1 In the program `myheatdisk.m` change the radiation coefficient  $d$  to `.01`. How does this affect the final temperature at  $r = 0$ ? How does it affect the rate of convergence of the method? Turn in both plots, the derivation, and answers to the questions.

# Lecture 36

## Parabolic PDEs - Explicit Method

### Heat Flow and Diffusion

The conduction of heat and diffusion of a chemical happen to be modeled by the same differential equation. The reason for this is that they both involve similar processes. Heat conduction occurs when hot, fast moving molecules bump into slower molecules and transfer some of their energy. In a solid this involves moles of molecules all moving in different, nearly random ways, but the net effect is that the energy eventually spreads itself out over a larger region. The diffusion of a chemical in a gas or liquid similarly involves large numbers of molecules moving in different, nearly random ways. These molecules eventually spread out over a larger region.

In three dimensions, the equation that governs both of these processes is the heat/diffusion equation

$$u_t = c\Delta u,$$

where  $c$  is the coefficient of conduction or diffusion, and

$$\Delta u(x, y, z) = u_{xx} + u_{yy} + u_{zz}.$$

The symbol  $\Delta$  in this context is called the *Laplacian*. If there is also a heat/chemical source, then it is incorporated a function  $g(x, y, z, t)$  in the equation as

$$u_t = c\Delta u + g.$$

In some problems the  $z$  dimension is irrelevant, either because the object in question is very thin, or  $u$  does not change in the  $z$  direction. In this case the equation is

$$u_t = c\Delta u = c(u_{xx} + u_{yy}).$$

Finally, in some cases only the  $x$  direction matters. In this case the equation is just

$$u_t = cu_{xx}, \tag{36.1}$$

or

$$u_t = cu_{xx} + g(x, t), \tag{36.2}$$

if there is a heat source represented by a function  $g(x, t)$ .

In this lecture we will learn a straight-forward technique for solving (36.1) and (36.2). It is very similar to the finite difference method we used for nonlinear boundary value problem in the previous lecture.

It is worth mentioning a related equation

$$u_t = c\Delta(u^\gamma) \quad \text{for } \gamma > 1,$$

which is called the porous-media equation. This equation models diffusion in a solid, but porous, material, such as sandstone or an earthen structure. We will not solve this equation numerically, but the methods introduced here would work. Many equations that involve 1 time derivative and 2 spatial derivatives are **parabolic** and the methods introduced in this lecture will work for most of them.

## Explicit Method Finite Differences

The one dimensional heat/diffusion equation  $u_t = cu_{xx}$ , has two independent variables,  $t$  and  $x$ , and so we have to discretize both. Since we are considering  $0 \leq x \leq L$ , we subdivide  $[0, L]$  into  $m$  equal subintervals, i.e. let

$$h = L/m$$

and

$$(x_0, x_1, x_2, \dots, x_{m-1}, x_m) = (0, h, 2h, \dots, L - h, L).$$

Similarly, if we are interested in solving the equation on an interval of time  $[0, T]$ , let

$$k = T/n$$

and

$$(t_0, t_1, t_2, \dots, t_{n-1}, t_n) = (0, k, 2k, \dots, T - k, T).$$

We will then denote the approximate solution at the grid points by

$$u_{ij} \approx u(x_i, t_j).$$

The equation  $u_t = cu_{xx}$  can then be replaced by the difference equations

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{c}{h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}). \quad (36.3)$$

Here we have used the forward difference for  $u_t$  and the central difference for  $u_{xx}$ . This equation can be solved for  $u_{i,j+1}$  to produce

$$u_{i,j+1} = ru_{i-1,j} + (1 - 2r)u_{i,j} + ru_{i+1,j} \quad (36.4)$$

for  $1 \leq i \leq m - 1$ ,  $0 \leq j \leq n - 1$ , where

$$r = \frac{ck}{h^2}. \quad (36.5)$$

The formula (36.4) allows us to calculate all the values of  $u$  at step  $j + 1$  using the values at step  $j$ .

Notice that  $u_{i,j+1}$  depends on  $u_{i,j}$ ,  $u_{i-1,j}$  and  $u_{i+1,j}$ . That is  $u$  at grid point  $i$  depends on its previous value and the values of its two nearest neighbors at the previous step (see Figure 36.1).

## Boundary and Initial Conditions

To solve the partial differential equation (36.1) or (36.2) we need boundary conditions. Just as in the previous section we will have to specify something about the ends of the domain, i.e. at  $x = 0$  and  $x = L$ . Two of the possibilities are just as they were for the steady state equation, fixed boundary conditions and insulated boundary conditions. We can implement both of those just as we did for the ODE boundary value problem.

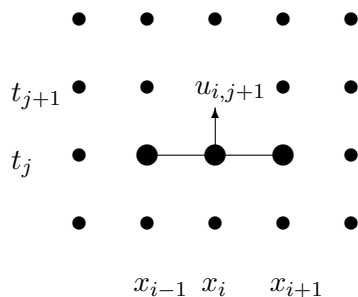


Figure 36.1: The value at grid point  $i$  depends on its previous values and the previous values of its nearest neighbors.

At third possibility is called **variable boundary conditions**. This is represented by time-dependent functions,

$$u(0, t) = g_1(t) \quad \text{and} \quad u(L, t) = g_2(t).$$

In a heat problem,  $g_1$  and  $g_2$  would represent heating or cooling applied to the ends. These are easily implemented in a program by letting  $u_{0,j} = g_1(t_j)$  and  $u_{m,j} = g_2(t_j)$ .

In addition to boundary conditions, we need an initial condition for this problem. This represents the state of the system when we begin, i.e. the initial temperature distribution or initial concentration profile. This is represented by

$$u(x, 0) = f(x).$$

To implement this in a program we let

$$u_{i,0} = f(x_i).$$

## Implementation

The following program (also available on the web page) implements the explicit method. It incorporates variable boundary conditions at both ends. To run it you must define functions  $f$ ,  $g_1$  and  $g_2$ . Notice that the main loop has only one line. The values of  $u$  are kept as a matrix. It is often convenient to define a matrix of the right dimension containing all zeros, and then fill in the calculated values as the program runs.

Run this program using  $L = 2$ ,  $T = 20$ ,  $f(x) = .5x$ ,  $g_1(t) = 0$ , and  $g_2(t) = \cos(t)$ . Note that  $g_1(t)$  must be input as `g1 = inline('0*t')`.

```

function [t x u] = myheat(f,g1,g2,L,T,m,n,c)
% function [t x u] = myheat(f,g1,g2,L,T,m,n,c)
% solve u_t = c u_xx for 0<=x<=L, 0<=t<=T
% BC: u(0, t) = g1(t); u(L,t) = g2(t)
% IC: u(x, 0) = f(x)
% Inputs:
% f -- inline function for IC
% g1,g2 -- inline functions for BC
% L -- length of rod
% T -- length of time interval
% m -- number of subintervals for x
% n -- number of subintervals for t
% c -- rate constant in equation
% Outputs:
% t -- vector of time points
% x -- vector of x points
% u -- matrix of the solution, u(i,j)~u(x(i),t(j))

h = L/m; k = T/n;
r = c*k/h^2; rr = 1 - 2*r;
x = linspace(0,L,m+1);
t = linspace(0,T,n+1);

%Set up the matrix for u:
u = zeros(m+1,n+1);

% assign initial conditions
u(:,1) = f(x);

% assign boundary conditions
u(1,:) = g1(t); u(m+1,:) = g2(t);

% find solution at remaining time steps
for j = 1:n
    u(2:m,j+1) = r*u(1:m-1,j) + rr*u(2:m,j) + r*u(3:m+1,j);
end

% plot the results
mesh(x,t,u')

```

## Exercises

- 36.1 Modify the program `myheat.m` to have an insulated boundary at  $x = L$  (rather than  $u(L, t) = g_2(t)$ ). Run the program with  $L = 2\pi$ ,  $T = 20$ ,  $c = .5$ ,  $g_1(t) = \sin(t)$  and  $f(x) = -\sin(x/4)$ . Experiment with  $m$  and  $n$ . Get a plot when the program is stable and one when it isn't. Turn in your program and plots.

# Lecture 37

## Solution Instability for the Explicit Method

As we saw in experiments using `myheat.m`, the solution can become unbounded unless the time steps are small. In this lecture we consider why.

### Writing the Difference Equations in Matrix Form

If we use the boundary conditions  $u(0) = u(L) = 0$  then the explicit method of the previous section has the form:

$$u_{i,j+1} = ru_{i-1,j} + (1 - 2r)u_{i,j} + ru_{i+1,j}, \quad 1 \leq i \leq m - 1, \quad 0 \leq j \leq n - 1, \quad (37.1)$$

where  $u_{0,j} = 0$  and  $u_{m,j} = 0$ . This is equivalent to the matrix equation

$$\mathbf{u}_{j+1} = A\mathbf{u}_j, \quad (37.2)$$

where  $\mathbf{u}_j$  is the column vector  $(u_{1,j}, u_{2,j}, \dots, u_{m,j})'$  representing the state at the  $j$ th time step and  $A$  is the matrix

$$A = \begin{pmatrix} 1 - 2r & r & 0 & \cdots & 0 \\ r & 1 - 2r & r & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & r & 1 - 2r & r \\ 0 & \cdots & 0 & r & 1 - 2r \end{pmatrix}. \quad (37.3)$$

Unfortunately, this matrix can have a property which is very bad in this context. Namely, it can cause exponential growth of error unless  $r$  is small. To see how this happens, suppose that  $\mathbf{U}_j$  is the vector of correct values of  $u$  at time step  $t_j$  and  $\mathbf{E}_j$  is the error of the approximation  $\mathbf{u}_j$ , then

$$\mathbf{u}_j = \mathbf{U}_j + \mathbf{E}_j.$$

From (37.2), the approximation at the next time step will be

$$\mathbf{u}_{j+1} = A\mathbf{U}_j + A\mathbf{E}_j,$$

and if we continue for  $k$  steps,

$$\mathbf{u}_{j+k} = A^k\mathbf{U}_j + A^k\mathbf{E}_j.$$

The problem with this is the term  $A^k\mathbf{E}_j$ . This term is exactly what we would do in the power method for finding the eigenvalue of  $A$  with the largest absolute value. If the matrix  $A$  has **ew**'s with absolute value greater than 1, then this term will grow exponentially. Figure 37.1 shows the largest absolute value of an **ew** of  $A$  as a function of the parameter  $r$  for various sizes of the matrix  $A$ . As you can see, for  $r > 1/2$  the largest absolute **ew** grows rapidly for any  $m$  and quickly becomes greater than 1.

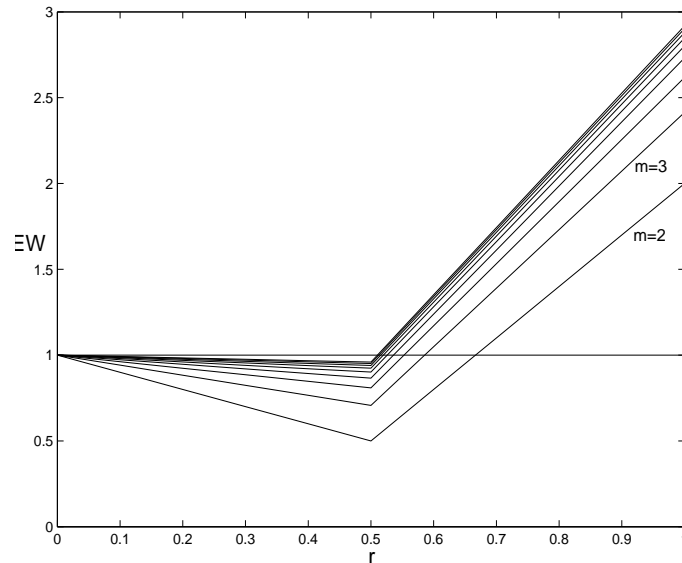


Figure 37.1: Maximum absolute eigenvalue  $EW$  as a function of  $r$  for the matrix  $A$  from the explicit method for the heat equation calculated for matrices  $A$  of sizes  $m = 2 \dots 10$ . Whenever  $EW > 1$  the method is unstable, i.e. errors grow exponentially with each step. When using the explicit method  $r < 1/2$  is a safe choice.

## Consequences

Recall that  $r = ck/h^2$ . Since this must be less than  $1/2$ , we have

$$k < \frac{h^2}{2c}.$$

The first consequence is obvious:  $k$  must be relatively small. The second is that  $h$  cannot be too small. Since  $h^2$  appears in the formula, making  $h$  small would force  $k$  to be extremely small!

A third consequence is that we have a converse of this analysis. Suppose  $r < .5$ . Then all the eigenvalues will be less than one. Recall that the error terms satisfy

$$\mathbf{u}_{j+k} = A^k \mathbf{U}_j + A^k \mathbf{E}_j.$$

If all the eigenvalues of  $A$  are less than 1 in absolute value then  $A^k \mathbf{E}_j$  grows smaller and smaller as  $k$  increases. This is really good. Rather than building up, the effect of any error diminishes as time passes! From this we arrive at the following principle: **If the explicit numerical solution for a parabolic equation does not blow up, then errors from previous steps fade away!**

Finally, we note that if we have other boundary conditions then instead of equation (37.2) we have

$$\mathbf{u}_{j+1} = A\mathbf{u}_j + r\mathbf{b}_j, \quad (37.4)$$

where the first and last entries of  $\mathbf{b}_j$  contain the boundary conditions and all the other entries are zero. In this case the errors behave just as before, if  $r > 1/2$  then the errors grow and if  $r < 1/2$  the errors fade away.

We can write a function program `myexppmatrix` that produces the matrix  $A$  in (37.3), for given inputs  $m$  and  $r$ . Without using loops we can use the `diag` command to set up the matrix:

```
function A = myexppmatrix(m,r)
% produces the matrix for the explicit method for a parabolic equation
% Inputs: m -- the size of the matrix
%         r -- the main parameter, ck/h^2
% Output: A -- an m by m matrix
u = (1-2*r)*ones(m,1);
v = r*ones(m-1,1);
A = diag(u) + diag(v,1) + diag(v,-1);
```

Test this using  $m = 6$  and  $r = .4, .6$ . Check the eigenvalues and eigenvectors of the resulting matrices:

```
> A = myexppmatrix(6,.6)
> [v e] = eig(A)
```

What is the “mode” represented by the eigenvector with the largest absolute eigenvalue? How is that reflected in the unstable solutions?

## Exercises

- 37.1 Let  $L = \pi$ ,  $T = 20$ ,  $f(x) = .1 \sin(x)$ ,  $g_1(t) = 0$ ,  $g_2(t) = 0$ ,  $c = .5$ , and  $m = 20$ , as used in the program `myheat.m`. Calculate the value of  $n$  corresponding to  $r = 1/2$ . Try different  $n$  in `myheat.m` to find precisely when the method works and when it fails. Is  $r = 1/2$  the boundary between failure and success? Hand in a plot of the last success and the first failure.
- 37.2 Write a script program that produces the graph in Figure 37.1 for  $m = 4$ . Your program should: define  $r$  values from 0 to 1, for each  $r$  create the matrix  $A$  using `myexppmatrix`, calculate the eigenvalues of  $A$ , find the max of the absolute values, and plot these numbers versus  $r$ .

# Lecture 38

## Implicit Methods

### The Implicit Difference Equations

By approximating  $u_{xx}$  and  $u_t$  at  $t_{j+1}$  rather than  $t_j$ , and using a backwards difference for  $u_t$ , the equation  $u_t = cu_{xx}$  is approximated by

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{c}{h^2}(u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}). \quad (38.1)$$

Note that all the terms have index  $j + 1$  except one and isolating this term leads to

$$u_{i,j} = -ru_{i-1,j+1} + (1 + 2r)u_{i,j+1} - ru_{i+1,j+1} \quad \text{for } 1 \leq i \leq m - 1, \quad (38.2)$$

where  $r = ck/h^2$  as before.

Now we have  $\mathbf{u}_j$  given in terms of  $\mathbf{u}_{j+1}$ . This would seem like a problem, until you consider that the relationship is linear. Using matrix notation, we have

$$\mathbf{u}_j = B\mathbf{u}_{j+1} - r\mathbf{b}_{j+1},$$

where  $\mathbf{b}_{j+1}$  represents the boundary condition. Thus to find  $\mathbf{u}_{j+1}$ , we need only solve the linear system

$$B\mathbf{u}_{j+1} = \mathbf{u}_j + r\mathbf{b}_{j+1}, \quad (38.3)$$

where  $\mathbf{u}_j$  and  $\mathbf{b}_{j+1}$  are given and

$$B = \begin{pmatrix} 1 + 2r & -r & & & & & \\ -r & 1 + 2r & -r & & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -r & 1 + 2r & -r & \\ & & & & -r & 1 + 2r & \end{pmatrix}. \quad (38.4)$$

Using this scheme is called the **implicit method** since  $\mathbf{u}_{j+1}$  is defined implicitly.

Since we are solving (38.1), the most important quantity is the maximum absolute eigenvalue of  $B^{-1}$ , which is 1 divided by the smallest  $\mathbf{e}\mathbf{w}$  of  $B$ . Figure 38.1 shows the maximum absolute  $\mathbf{e}\mathbf{w}$ 's of  $B^{-1}$  as a function of  $r$  for various size matrices. Notice that this absolute maximum is always less than 1. Thus errors are always diminished over time and so this method is always stable. For the same reason it is also always as accurate as the individual steps.

Both this implicit method and the explicit method in the previous lecture make  $O(h^2)$  error in approximating  $u_{xx}$  and  $O(k)$  error in approximating  $u_t$ , so they have total error  $O(h^2 + k)$ . Thus although the stability condition allows the implicit method to use arbitrarily large  $k$ , to maintain accuracy we still need  $k \sim h^2$ .

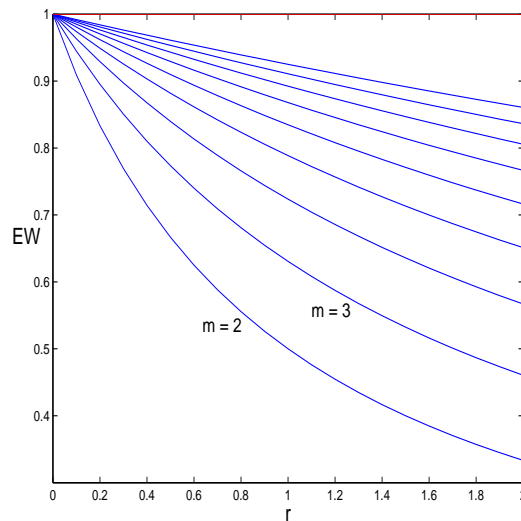


Figure 38.1: Maximum absolute eigenvalue  $EW$  as a function of  $r$  for the matrix  $B^{-1}$  from the implicit method for the heat equation calculated for matrices  $B$  of sizes  $m = 2 \dots 10$ . Whenever  $EW < 1$  the method is stable, i.e. it is always stable.

## Crank-Nicholson Method

Now that we have two different methods for solving parabolic equation, it is natural to ask, “can we improve by taking an average of the two methods?” The answer is yes.

We implement a weighted average of the two methods by considering an average of the approximations of  $u_{xx}$  at  $j$  and  $j + 1$ . This leads to the equations

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{\lambda c}{h^2}(u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}) + \frac{(1-\lambda)c}{h^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}). \quad (38.5)$$

The implicit method contained in these equations is called the **Crank-Nicholson method**. Gathering terms yields the equations

$$-r\lambda u_{i-1,j+1} + (1 + 2r\lambda)u_{i,j+1} - r\lambda u_{i+1,j+1} = r(1-\lambda)u_{i-1,j} + (1 - 2r(1-\lambda))u_{i,j} + r(1-\lambda)u_{i+1,j}.$$

In matrix notation this is

$$B_\lambda \mathbf{u}_{j+1} = A_\lambda \mathbf{u}_j + r\mathbf{b}_{j+1},$$

where

$$A_\lambda = \begin{pmatrix} 1 - 2(1-\lambda)r & (1-\lambda)r & & & \\ (1-\lambda)r & 1 - 2(1-\lambda)r & (1-\lambda)r & & \\ & \ddots & \ddots & \ddots & \\ & & (1-\lambda)r & 1 - 2(1-\lambda)r & (1-\lambda)r \\ & & & (1-\lambda)r & 1 - 2(1-\lambda)r \end{pmatrix} \quad (38.6)$$

and

$$B_\lambda = \begin{pmatrix} 1+2r\lambda & -r\lambda & & & & \\ -r\lambda & 1+2r\lambda & -r\lambda & & & \\ & & \ddots & \ddots & \ddots & \\ & & & -r\lambda & 1+2r\lambda & -r\lambda \\ & & & & -r\lambda & 1+2r\lambda \end{pmatrix}. \quad (38.7)$$

In this equation  $\mathbf{u}_j$  and  $\mathbf{b}_{j+1}$  are known,  $\mathbf{A}\mathbf{u}_j$  can be calculated directly, and then the equation is solved for  $\mathbf{u}_{j+1}$ .

If we choose  $\lambda = 1/2$ , then we are in effect doing a central difference for  $u_t$ , which has error  $O(k^2)$ . Our total error is then  $O(h^2 + k^2)$ . With a bit of work, we can show that the method is always stable, and so we can use  $k \sim h$  without a problem.

To get optimal accuracy with a weighted average, it is always necessary to use the right weights. For the Crank-Nicholson method with a given  $r$ , we need to choose

$$\lambda = \frac{r - 1/6}{2r}.$$

This choice will make the method have truncation error of order  $O(h^4 + k^2)$ , which is really good considering that the implicit and explicit methods each have truncation errors of order  $O(h^2 + k)$ . Surprisingly, we can do even better if we also require

$$r = \frac{\sqrt{5}}{10} \approx 0.22361,$$

and, consequently,

$$\lambda = \frac{3 - \sqrt{5}}{6} \approx 0.12732.$$

With these choices, the method has truncation error of order  $O(h^6)$ , which is absolutely amazing.

To appreciate the implications, suppose that we need to solve a problem with 4 significant digits. If we use the explicit or implicit method alone then we will need  $h^2 \approx k \approx 10^{-4}$ . If  $L = 1$  and  $T \approx 1$ , then we need  $m \approx 100$  and  $n \approx 10,000$ . Thus we would have a total of 1,000,000 grid points, almost all on the interior. This is a lot.

Next suppose we solve the same problem using the optimal Crank-Nicholson method. We would need  $h^6 \approx 10^{-4}$  which would require us to take  $m \approx 4.64$ , so we would take  $m = 5$  and have  $h = 1/5$ . For  $k$  we need  $k = (\sqrt{5}/10)h^2/c$ . If  $c = 1$ , this gives  $k = \sqrt{5}/250 \approx 0.0089442$  so we would need  $n \approx 112$  to get  $T \approx 1$ . This gives us a total of 560 interior grid points, or, a factor of 1785 fewer than the explicit or implicit method alone.

## Exercises

- 38.1 Modify the program `myexppmatrix` from exercise 36.2 into a function program `myimpmatrix` that produces the matrix  $B_\lambda$ , for given inputs  $m$ ,  $r$  and  $\lambda$ . Modify your script from exercise 36.2 to use  $B_\lambda^{-1}$  for  $\lambda = 1/2$  and  $m = 4$ . It should produce a graph similar to that in Figure 38.1 for  $m = 4$ . Turn in the programs and the plot.

# Lecture 39

## Finite Difference Method for Elliptic PDEs

### Examples of Elliptic PDEs

**Elliptic** PDE's are equations with second derivatives in space and no time derivative. The most important examples are Laplace's equation

$$\Delta u = u_{xx} + u_{yy} + u_{zz} = 0$$

and the Poisson equation

$$\Delta u = f(x, y, z).$$

These equations are used in a large variety of physical situations such as: steady state heat problems, steady state chemical distributions, electrostatic potentials, elastic deformation and steady state fluid flows.

For the sake of clarity we will only consider the two dimensional problem. A good model problem in this dimension is the elastic deflection of a membrane. Suppose that a membrane such as a sheet of rubber is stretched across a rectangular frame. If some of the edges of the frame are bent, or if forces are applied to the sheet then it will deflect by an amount  $u(x, y)$  at each point  $(x, y)$ . This  $u$  will satisfy the boundary value problem:

$$\begin{aligned} u_{xx} + u_{yy} &= f(x, y) & \text{for } (x, y) \text{ in } R, \\ u(x, y) &= g(x, y) & \text{for } (x, y) \text{ on } \partial R, \end{aligned} \tag{39.1}$$

where  $R$  is the rectangle,  $\partial R$  is the edge of the rectangle,  $f(x, y)$  is the force density (pressure) applied at each point and  $g(x, y)$  is the deflection at the edge.

### The Finite Difference Equations

Suppose the rectangle is described by

$$R = \{a \leq x \leq b, c \leq y \leq d\}.$$

We will divide  $R$  in subrectangles. If we have  $m$  subdivisions in the  $x$  direction and  $n$  subdivisions in the  $y$  direction, then the step size in the  $x$  and  $y$  directions respectively are

$$h = \frac{b-a}{m} \quad \text{and} \quad k = \frac{d-c}{n}.$$

We obtain the finite difference equations for (39.1) by replacing  $u_{xx}$  and  $u_{yy}$  by their central differences to obtain

$$\frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{k^2} = f(x_i, y_j) = f_{ij} \quad (39.2)$$

for  $1 \leq i \leq m-1$  and  $1 \leq j \leq n-1$ . The boundary conditions are introduced by:

$$u_{0,j} = g(a, y_j), \quad u_{m,j} = g(b, y_j), \quad u_{i,0} = g(x_i, c), \quad \text{and} \quad u_{i,n} = g(x_i, d). \quad (39.3)$$

## Direct Solution of the Equations

Notice that since the edge values are prescribed, there are  $(m-1) \times (n-1)$  grid points where we need to determine the solution. Note also that there are exactly  $(m-1) \times (n-1)$  equations in (39.2). Finally, notice that the equations are all linear. Thus we could solve the equations exactly using matrix methods. To do this we would first need to express the  $u_{ij}$ 's as a vector, rather than a matrix. To do this there is a standard procedure: let  $\mathbf{u}$  be the column vector we get by placing one column after another from the columns of  $(u_{ij})$ . Thus we would list  $u_{.,1}$  first then  $u_{.,2}$ , etc.. Next we would need to write the matrix  $A$  that contains the coefficients of the equations (39.2) and incorporate the boundary conditions in a vector  $\mathbf{b}$ . Then we could solve an equation of the form

$$A\mathbf{u} = \mathbf{b}. \quad (39.4)$$

Setting up and solving this equation is called the direct method.

An advantage of the direct method is that solving (39.4) can be done relatively quickly and accurately. The drawback of the direct method is that one must set up  $\mathbf{u}$ ,  $A$  and  $\mathbf{b}$ , which is confusing. Further, the matrix  $A$  has dimensions  $(m-1)(n-1) \times (m-1)(n-1)$ , which can be rather large. Although  $A$  is large, many of its elements are zero. Such a matrix is called *sparse* and there are special methods intended for efficiently working with sparse matrices.

## Iterative Solution

A usually preferred alternative to the direct method described above is to solve the finite difference equations iteratively. To do this, first solve (39.2) for  $u_{ij}$ , which yields

$$u_{ij} = \frac{1}{2(h^2 + k^2)} (k^2(u_{i+1,j} + u_{i-1,j}) + h^2(u_{i,j+1} + u_{i,j-1}) - h^2k^2f_{ij}). \quad (39.5)$$

This method is another example of a *relaxation method*. Using this formula, along with (39.3), we can update  $u_{ij}$  from its neighbors, just as we did in the relaxation method for the nonlinear boundary value problem. If this method converges, then the result is an approximate solution.

The iterative solution is implemented in the program `mypoissson.m` on the class web site. Download and read it. You will notice that `maxit` is set to 0. Thus the program will not do any iteration, but will plot the initial guess. The initial guess in this case consists of the proper boundary values at the edges, and zero everywhere in the interior. To see the solution evolve, gradually increase `maxit`.

## Exercises

39.1 Modify the program `mypoisson.m` (from the class web page) in the following ways:

1. Change  $x = b$  to be an insulated boundary, i.e.  $u_x(b, y) = 0$ .
2. Change the force  $f(x, y)$  to a negative constant  $-p$ .

Experiment with various values of  $p$  and `maxit`. Obtain plots for small and large  $p$ . Tell how many iterations are needed for convergence in each. For large  $p$  plot also a couple of intermediate steps.

# Lecture 40

## Convection-Diffusion Equations

### Exercises

40.1

# Lecture 41

## Finite Elements\*

### Triangulating a Region

A disadvantage of finite difference methods is that they require a very regular grid, and thus a very regular region, either rectangular or a regular part of a rectangle. Finite elements is a method that works for any shape region because it is not built of a grid, but on triangulation of the region, i.e. cutting the region up into triangles as we did in a previous lecture.

The following figure shows a triangularization of a region.

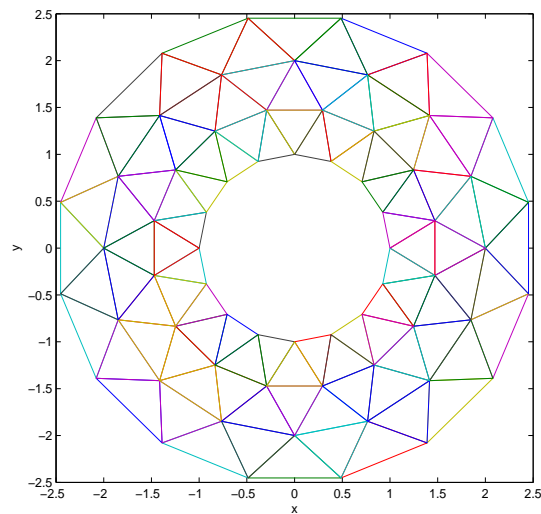


Figure 41.1: An annular region with a triangulation. Notice that the nodes and triangles are very evenly spaced.

This figure was produced by the script program `mywasher.m`. Notice that the nodes are evenly distributed. This is good for the finite element process where we will use it.

Open the program `mywasher.m`. This program defines a triangulation by defining the vertices in a matrix `V` in which each row contains the  $x$  and  $y$  coordinates of a vertex. Notice that we list the interior nodes first, then the boundary nodes.

Triangles are defined in the matrix `T`. Each row of `T` has three integer numbers indicating the indices of the nodes that form a triangle. For instance the first row is `43 42 25`, so  $T_1$  is the triangle

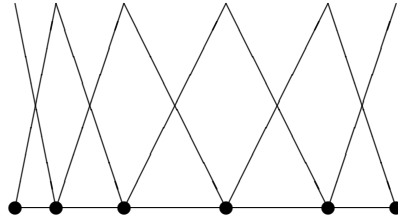


Figure 41.2: The finite elements for the triangulation of a one dimensional object.

with vertices  $\mathbf{v}_{43}$ ,  $\mathbf{v}_{42}$  and  $\mathbf{v}_{25}$ . The matrix  $T$  in this case was produced by the MATLAB command `delaunay`. The command produced more triangles than desire and the unwanted ones were deleted.

We a three dimensional plot of the region and triangles is produced by having the last line be `trimesh(T,x,y,z)`.

## What is a finite element?

The finite element method is a mathematically complicated process. However, a finite element is actually a very simple object.

An element is a piecewise-defined function  $\Phi_j$  which is:

- Linear (or some other simple function) on each triangle.
- There is exactly one element corresponding each node.
- An element equals 1 at the node  $j$  and 0 at all other nodes.

If there are  $n$  vertices's, then there are also  $n$  elements. We will use the index  $j$  to number both vertices's and the corresponding elements. Thus the set of elements is

$$\{\Phi_j\}_{j=1}^n.$$

As we have defined it, an element,  $\Phi_j$ , is a function whose graph is a pyramid with its peak over node  $j$ .

If we consider one dimension, then a triangulation is just a subdivision into subintervals. In Figure 41.2 we show an uneven subdivision of an interval and the elements corresponding to each node.

## What is a finite element solution?

A finite element (approximate) solution is a linear combination of the elements:

$$U = \sum_{j=1}^n C_j \Phi_j. \quad (41.1)$$

Thus finding a finite element solution amounts to finding the best values for the constants  $\{C_j\}_{j=1}^n$ .

In the program `mywasher.m`, the vector  $\mathbf{c}$  contains the node values  $C_j$ . These values give the height at each node in the graph. For instance if we set all equal to 0 except one equal to 1, then the function is a finite element. Do this for one boundary node, then for one interior node.

Notice that a sum of linear functions is a linear function. Thus the solution using linear elements is a piecewise linear function. Also notice that if we denote the  $j$ -th vertex by  $\mathbf{v}_j$ , then

$$U(\mathbf{v}_j) = C_j. \quad (41.2)$$

Thus we see that **the constants  $C_j$  are just the values at the nodes.**

Take the one-dimensional case. Since we know that the solution is linear on each subinterval, knowing the values at the endpoints of the subintervals, i.e. the nodes, gives us complete knowledge of the solution. Figure 41.3 could be a finite element solution since it is piecewise linear.

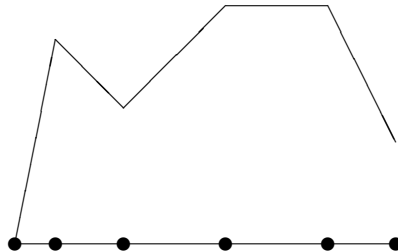


Figure 41.3: A possible finite element solution for a one dimensional object. Values are assigned at each node and a linear interpolant is used in between.

In the two-dimensional case, the solution is linear on each triangle, and so again, if we know the values  $\{C_j\}_{j=1}^n$  at the nodes then we know everything.

## Experiment with finite elements

By changing the values in  $\mathbf{c}$  in the program we can produce different three dimensional shapes based on the triangles. The point then of a finite element solution is to find the values at the nodes that best approximate the true solution. This task can be subdivided into two parts: (1) assigning the values at the boundary nodes and (2) assigning the values at the interior nodes.

## Values at boundary nodes

Once a triangulation and a set of finite elements is set up, the next step is to incorporate the boundary conditions of the problem. Suppose that we have fixed boundary conditions, i.e. of the form

$$u(\mathbf{x}) = g(\mathbf{x}), \quad \text{for } \mathbf{x} \in \partial D,$$

where  $D$  is the object (domain) and  $\partial D$  is its boundary. Then the boundary condition directly determines the values on the boundary nodes.

In particular, suppose that  $\mathbf{v}_\ell$  is a boundary node. Since  $\Phi_\ell(\mathbf{v}_\ell) = 1$  and all the other elements are zero at node  $\mathbf{v}_\ell$ , then to make

$$U(\mathbf{v}_\ell) = \sum_{j=1}^n C_j \Phi_j(\mathbf{v}_\ell) = g(\mathbf{v}_\ell),$$

we must choose

$$C_\ell = g(\mathbf{v}_\ell).$$

Thus the constants  $C_j$  for the boundary nodes are set at exactly the value of the boundary condition at the nodes.

Thus, if there are  $m$  interior nodes, then

$$C_j = g(\mathbf{v}_j), \quad \text{for all } m+1 \leq j \leq n.$$

In the program `mywasher` the first 32 vertices correspond to interior nodes and the last 32 correspond to boundary nodes. By setting the last 32 values of  $\mathbf{z}$ , we achieve the boundary conditions. We could do this by adding the following commands to the program:

```
z(33:64) = .5;
```

or more elaborately we might use functions:

```
z(33:48) = x(33:48).^2 - .5*y(33:48).^2;
```

```
z(49:64) = .2*cos(y(49:64));
```

## Exercises

- 41.1 Generate an interesting or useful 2-d object and a well-distributed triangulation of it. Plot the region. Plot one interior finite element and one boundary finite element. Assign values to the boundary using a function (or functions) and plot the region with the boundary values.

# Lecture 42

## Determining Internal Node Values\*

As seen in the previous section, a finite element solution of a boundary value problem boils down to finding the best values of the constants  $\{C_j\}_{j=1}^n$ , which are the values of the solution at the nodes. The interior nodes values are determined by *variational principles*. Variational principles usually amount to **minimizing internal energy**. It is a physical principle that systems seek to be in a state of minimal energy and this principle is used to find the internal node values.

### Variational Principles

For the differential equations that describe many physical systems, the internal energy of the system is an integral. For instance, for the steady state heat equation

$$u_{xx} + u_{yy} = g(x, y) \quad (42.1)$$

the internal energy is the integral

$$I[u] = \iint_R u_x^2 + u_y^2 + 2g(x, y)u(x, y) dA, \quad (42.2)$$

where  $R$  is the region on which we are working. It can be shown that  $u(x, y)$  is a solution of (42.1) if and only if it is minimizer of  $I[u]$  in (42.2).

### The finite element solution

Recall that a finite element solution is a linear combination of finite element functions:

$$U(x, y) = \sum_{j=1}^n C_j \Phi_j(x, y),$$

where  $n$  is the number of nodes. To obtain the values at the internal nodes, we will plug  $U(x, y)$  into the energy integral and minimize. That is, we find the minimum of

$$I[U]$$

for all choices of  $\{C_j\}_{j=1}^m$ , where  $m$  is the number of internal nodes. In this as with any other minimization problem, the way to find a possible minimum is to differentiate the quantity with respect to the variables and set the results to zero. In this case the free variables are  $\{C_j\}_{j=1}^m$ . Thus to find the minimizer we should try to solve

$$\frac{\partial I[U]}{\partial C_j} = 0 \quad \text{for } 1 \leq j \leq m. \quad (42.3)$$

We call this set of equations the **internal node equations**. At this point we should ask whether the internal node equations can be solved, and if so, is the solution actually a minimizer (and not a maximizer). The following two facts answer these questions. These facts make the finite element method practical:

- **for most applications the internal node equations are linear**
- **for most applications the internal node equations give a minimizer**

We can demonstrate the first fact using an example.

### Application to the steady state heat equation

If we plug the candidate finite element solution  $U(x, y)$  into the energy integral for the heat equation (42.2), we obtain

$$I[U] = \iint_R U_x(x, y)^2 + U_y(x, y)^2 + 2g(x, y)U(x, y) dA. \quad (42.4)$$

Differentiating with respect to  $C_j$  we obtain the internal node equations

$$0 = \iint_R 2U_x \frac{\partial U_x}{\partial C_j} + 2U_y \frac{\partial U_y}{\partial C_j} + 2g(x, y) \frac{\partial U}{\partial C_j} dA \quad \text{for } 1 \leq j \leq m. \quad (42.5)$$

Now we have several simplifications. First note that since

$$U(x, y) = \sum_{j=1}^n C_j \Phi_j(x, y),$$

we have

$$\frac{\partial U}{\partial C_j} = \Phi_j(x, y).$$

Also note that

$$U_x(x, y) = \sum_{j=1}^n C_j \frac{\partial}{\partial x} \Phi_j(x, y),$$

and so

$$\frac{\partial U_x}{\partial C_j} = (\Phi_j)_x.$$

Similarly,  $\frac{\partial U_y}{\partial C_j} = (\Phi_j)_y$ . The integral (42.5) then becomes

$$0 = 2 \iint U_x(\Phi_j)_x + U_y(\Phi_j)_y + g(x, y)\Phi_j(x, y) dA \quad \text{for } 1 \leq j \leq m.$$

Next we use the fact that the region  $R$  is subdivided into triangles  $\{T_i\}_{i=1}^p$  and the functions in question have different definitions on each triangle. The integral then is a sum of the integrals:

$$0 = 2 \sum_{i=1}^p \iint_{T_i} U_x(\Phi_j)_x + U_y(\Phi_j)_y + g(x, y)\Phi_j(x, y) dA \quad \text{for } 1 \leq j \leq m.$$

Now note that the function  $\Phi_j(x, y)$  is linear on triangle  $T_i$  and so

$$\Phi_{ij}(x, y) = \Phi_j|_{T_i}(x, y) = a_{ij} + b_{ij}x + c_{ij}y.$$

This gives us the following simplification:

$$(\Phi_{ij})_x(x, y) = b_{ij} \quad \text{and} \quad (\Phi_{ij})_y(x, y) = c_{ij}.$$

Also,  $U_x$  and  $U_y$  restricted to  $T_i$  have the form:

$$U_x = \sum_{k=1}^n C_k b_{ik} \quad \text{and} \quad U_y = \sum_{k=1}^n C_k c_{ik}.$$

The internal node equations then reduce to

$$0 = \sum_{i=1}^p \iint_{T_i} \left( \sum_{k=1}^n C_k b_{ik} \right) b_{ij} + \left( \sum_{k=1}^n C_k c_{ik} \right) c_{ij} + g(x, y) \Phi_{ij}(x, y) dA \quad \text{for } 1 \leq j \leq m.$$

Now notice that  $(\sum_{k=1}^n C_k b_{ik}) b_{ij}$  is just a constant on  $T_i$ , and, thus, we have

$$\iint_{T_i} \left( \sum_{k=1}^n C_k b_{ik} \right) b_{ij} + \left( \sum_{k=1}^n C_k c_{ik} \right) c_{ij} = \left[ \left( \sum_{k=1}^n C_k b_{ik} \right) b_{ij} + \left( \sum_{k=1}^n C_k c_{ik} \right) c_{ij} \right] A_i,$$

where  $A_i$  is just the area of  $T_i$ . Finally, we apply the Three Corners rule to make an approximation to the integral

$$\iint_{T_i} g(x, y) \Phi_{ij}(x, y) dA.$$

Since  $\Phi_{ij}(x_k, y_k) = 0$  if  $k \neq j$  and even  $\Phi_{ij}(x_j, y_j) = 0$  if  $T_i$  does not have a corner at  $(x_j, y_j)$ , we get the approximation

$$\Phi_{ij}(x_j, y_j) g(x_j, y_j) A_i / 3.$$

If  $T_i$  does have a corner at  $(x_j, y_j)$  then  $\Phi_{ij}(x_j, y_j) = 1$ .

Summarizing, the internal node equations are:

$$0 = \sum_{i=1}^p \left[ \left( \sum_{k=1}^n C_k b_{ik} \right) b_{ij} + \left( \sum_{k=1}^n C_k c_{ik} \right) c_{ij} + \frac{1}{3} g(x_j, y_j) \Phi_{ij}(x_j, y_j) \right] A_i \quad \text{for } 1 \leq j \leq m.$$

While not pretty, these equations are in fact linear in the unknowns  $\{C_j\}$ .

## Experiment

Download the program `myfiniteelem.m` from the class web site.

This program produces a finite element solution for the steady state heat equation without source term:

$$u_{xx} + u_{yy} = 0.$$

The boundary values are input directly on the line:

$$c = [ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 ];$$

Try different values for  $c$ . You will see that the program works no matter what you choose.

## Exercises

42.1 Study for the final!

# Review of Part IV

## Methods and Formulas

### Initial Value Problems

#### Reduction to First order system:

For an  $n$ -th order equation that can be solved for the  $n$ -th derivative:

$$x^{(n)} = f\left(t, x, \dot{x}, \ddot{x}, \dots, \frac{d^{n-1}x}{dt^{n-1}}\right). \quad (42.6)$$

Then use the standard change of variables:

$$\begin{aligned} y_1 &= x \\ y_2 &= \dot{x} \\ &\vdots \\ y_n &= x^{(n-1)} = \frac{d^{n-1}x}{dt^{n-1}}. \end{aligned} \quad (42.7)$$

Differentiating results in a first-order system:

$$\begin{aligned} \dot{y}_1 &= \dot{x} = y_2 \\ \dot{y}_2 &= \ddot{x} = y_3 \\ &\vdots \\ \dot{y}_n &= x^{(n)} = f(t, y_1, y_2, \dots, y_n). \end{aligned} \quad (42.8)$$

#### Euler's method:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + hf(t_i, \mathbf{y}_i).$$

#### Modified (or Improved) Euler method:

$$\begin{aligned} \mathbf{k}_1 &= hf(t_i, \mathbf{y}_i) \\ \mathbf{k}_2 &= hf(t_i + h, \mathbf{y}_i + \mathbf{k}_1) \\ \mathbf{y}_{i+1} &= \mathbf{y}_i + \frac{1}{2}(\mathbf{k}_1 + \mathbf{k}_2) \end{aligned}$$

### Boundary Value Problems

#### Finite Differences:

Replace the Differential Equation by Difference Equations on a grid.  
Review the lecture on Numerical Differentiation.

**Explicit Method Finite Differences for Parabolic PDE (heat):**

$$u_t \mapsto \frac{u_{i,j+1} - u_{ij}}{k} \quad u_{xx} \mapsto \frac{u_{i-1,j} - 2u_{ij} + u_{i+1,j}}{h^2} \quad (42.9)$$

leads to:

$$u_{i,j+1} = ru_{i-1,j} + (1 - 2r)u_{i,j} + ru_{i+1,j},$$

where  $h = L/m$ ,  $k = T/n$ , and  $r = ck/h^2$ . The stability condition is  $r < 1/2$ .

**Implicit Method Finite Differences for Parabolic PDE (heat):**

$$u_t \mapsto \frac{u_{i,j+1} - u_{ij}}{k} \quad u_{xx} \mapsto \frac{u_{i-1,j+1} - 2u_{i,j+1} + u_{i+1,j+1}}{h^2} \quad (42.10)$$

leads to:

$$u_{i,j} = -ru_{i-1,j+1} + (1 + 2r)u_{i,j+1} - ru_{i+1,j+1},$$

which is always stable and has truncation error  $O(h^2 + k)$ .

**Crank-Nicholson Method Finite Differences for Parabolic PDE (heat):**

$$-ru_{i-1,j+1} + 2(1 + r)u_{i,j+1} - ru_{i+1,j+1} = ru_{i-1,j} + 2(1 - r)u_{i,j} + ru_{i+1,j},$$

which is always stable and has truncation error  $O(h^2 + k^2)$  or better if  $r$  and  $\lambda$  are chosen optimally.

**Finite Difference Method for Elliptic PDEs:**

$$u_{xx} + u_{yy} = f(x, y) \mapsto \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{k^2} = f(x_i, y_j) = f_{ij},$$

**Finite Elements:**

Based on triangles instead of rectangles.

Can be used for irregularly shaped objects.

An element: Pyramid shaped function at a node.

A finite element solution is a linear combination of finite element functions:

$$U(x, y) = \sum_{j=1}^n C_j \Phi_j(x, y),$$

where  $n$  is the number of nodes, and where  $U$  is an approximation of the true solution.

$C_j$  is the value of the solution at node  $j$ .

$C_j$  at the boundary nodes are given by boundary conditions.

$C_j$  at interior nodes are determined by variation principles.

The last step in determining  $C_j$ 's is solving a linear system of equations.

**MATLAB**

```
> [T Y] = ode45(f, tspan, y0)
```

Initial value problem solver that uses the Runge-Kutta 45 method.

$f$  must be a vector valued function,  $y0$  is the initial vector.

RK 45 uses variable step size to control speed vs. accuracy.

