

Part III

Functions and Data

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

Lecture 19

Polynomial and Spline Interpolation

A Chemical Reaction

In a chemical reaction the concentration level y of the product at time t was measured every half hour. The following results were found:

t	0	.5	1.0	1.5	2.0
y	0	.19	.26	.29	.31

We can input this data into MATLAB as:

```
> t1 = 0:.5:2  
> y1 = [ 0 .19 .26 .29 .31 ]
```

and plot the data with:

```
> plot(t1,y1)
```

MATLAB automatically connects the data with line segments. This is the simplest form of *interpolation*, meaning fitting a graph (of a function) between data points. What if we want a smoother graph? Try:

```
> plot(t1,y1,'*')
```

which will produce just asterisks at the data points. Next click on **Tools**, then click on the **Basic Fitting** option. This should produce a small window with several fitting options. Begin clicking them one at a time, clicking them off before clicking the next. Which ones produce a good-looking fit? You should note that the spline, the shape-preserving interpolant and the 4th degree polynomial produce very good curves. The others do not. We will discuss polynomial interpolation and spline interpolation in this lecture.

Polynomial Fitting

The most general degree n polynomial is:

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

If we have exactly $n + 1$ data points, that is enough to exactly determine the $n + 1$ coefficients of $p_n(x)$ (as long as the data does not have repeated x values). If there are more data points, that would give us an overdetermined system (more equations than unknowns) and if there is less data the system would be undetermined.

Conversely, if we have n data points, then an $n - 1$ degree polynomial has exactly enough coefficients to fit the data. This is the case in the example above; there are 5 data points so there is exactly one 4th degree polynomial that fits the data.

When we tried to use a 5th or higher degree polynomial MATLAB returned a warning that the polynomial is not unique since “degree \geq number of data points”.

When we tried a lower degree polynomial, we did not get a warning, but we did notice that the resulting curve does not hit all the data points. If there was an overdetermined system how did MATLAB come up with a pretty good approximation (for cubic)? The answer is the Least Squares Method, which we will study later.

Predicting the future?

Suppose we want to use the data to extrapolate into the future. Set the plot to the 4th degree polynomial. Then click the **Edit** button and select the **Axes Properties** option. A box should appear that allows you to adjust the domain of the x axes. Change the upper limit of the x -axis from 2 to 4. Based on the 4th degree polynomial, what will the chemical reaction do in the future? Is this reasonable?

Next change from 4th degree polynomial to spline interpolant. According to the spline, what will the chemical reaction do in the future? Try the shape-preserving interpolant also.

From our (limited) knowledge of chemical reactions, what should be the behavior as time goes on? It should reach a limiting value (chemical equilibrium). Could we use the data to predict this equilibrium value? Yes, we could and it is done all the time in many different contexts, but to do so we need to know that there is an equilibrium to predict. This requires that we understand the chemistry of the problem. Thus we have the following principle: To *extrapolate* beyond the data, one must have some knowledge of the process.

More data

Generally one would think that more data is better. Input the following data vectors:

```
> t2 = [ 0 .1 .4 .5 .6 1.0 1.4 1.5 1.6 1.9 2.0]
```

```
> y2 = [ 0 .06 .17 .19 .21 .26 .29 .29 .30 .31 .31]
```

There are 11 data points, so a 10-th degree polynomial will fit the data. However, this does not give a good graph. Thus: **A polynomial fit is better for small data sets.**

Finally, we note that each data point (x_i, y_i) gives an equation:

$$p_n(x_i) = a_0 + a_1x_i + a_2x_i^2 + \cdots + a_nx_i^n = y_i.$$

The unknowns are the coefficients a_0, \dots, a_n . This equation is linear in these unknowns, so determining a polynomial fit requires solving a linear system of equations. This makes polynomial fits quick to calculate since *solving linear systems is what computers do best*.

A challenging data set

Input the following data set:

```
> x = -4:1:5
```

```
> y = [ 0 0 0 1 1 1 0 0 0 0]
```

and plot it:

```
> plot(x,y,'*')
```

There are 10 data points, so there is a unique 9 degree polynomial that fits the data. Under **Tools** and **Basic Fitting** select the 9th degree polynomial fit. How does it look? De-select the 9th degree polynomial and select the spline interpolant. This should produce a much more satisfactory graph and the shape-preserving spline should be even better. In this section we learn what a spline is.

The idea of a spline

The general idea of a spline is this: on each interval between data points, represent the graph with a simple function. The simplest spline is something very familiar to you; it is obtained by connecting the data with lines. Since linear is the most simple function of all, linear interpolation is the simplest form of spline. The next simplest function is quadratic. If we put a quadratic function on each interval then we should be able to make the graph a lot smoother. If we were really careful then we should be able to make the curve smooth at the data points themselves by matching up the derivatives. This can be done and the result is called a quadratic spline. Using cubic functions or 4th degree functions should be smoother still. So, where should we stop? There is an almost universal consensus that *cubic* is the optimal degree for splines and so we focus the rest of the lecture on cubic splines.

Cubic spline

Again, the basic idea of the cubic spline is that we represent the function by a different cubic function on each interval between data points. That is, if there are n data points, then the spline $S(x)$ is the function:

$$S(x) = \begin{cases} C_1(x), & x_0 \leq x \leq x_1 \\ C_i(x), & x_{i-1} \leq x \leq x_i \\ C_n(x), & x_{n-1} \leq x \leq x_n \end{cases} \quad (19.1)$$

where each C_i is a cubic function. The most general cubic function has the form:

$$C_i(x) = a_i + b_i x + c_i x^2 + d_i x^3.$$

To determine the spline we must determine the coefficients, a_i , b_i , c_i , and d_i for each i . Since there are n intervals, there are $4n$ coefficients to determine. First we require that the spline be exact at the data:

$$C_i(x_{i-1}) = y_{i-1} \quad \text{and} \quad C_i(x_i) = y_i, \quad (19.2)$$

at every data point. In other words,

$$a_i + b_i x_{i-1} + c_i x_{i-1}^2 + d_i x_{i-1}^3 = y_{i-1} \quad \text{and} \quad a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 = y_i.$$

Notice that there are $2n$ of these conditions. Then to make $S(x)$ as smooth as possible we require:

$$\begin{aligned} C'_i(x_i) &= C'_{i+1}(x_i) \\ C''_i(x_i) &= C''_{i+1}(x_i), \end{aligned} \quad (19.3)$$

at all the internal points, i.e. $x_1, x_2, x_3, \dots, x_{n-1}$. In terms of the points these conditions can be written as:

$$\begin{aligned} b_i + 2c_i x_i + 3d_i x_i^2 &= b_{i+1} + 2c_{i+1} x_i + 3d_{i+1} x_i^2 \\ 2c_i + 6d_i x_i &= 2c_{i+1} + 6d_{i+1} x_i. \end{aligned} \quad (19.4)$$

There are $2(n-1)$ of these conditions. Since each C_i is cubic, there are a total of $4n$ coefficients in the formula for $S(x)$. So far we have $4n-2$ equations, so we are 2 equations short of being able to determine all the coefficients. At this point we have to make a choice. The usual choice is to require:

$$C''_1(x_0) = C''_n(x_n) = 0. \quad (19.5)$$

These are called *natural* or *simple* boundary conditions. The other common option is called *clamped* boundary conditions:

$$C'_1(x_0) = C'_n(x_n) = 0. \quad (19.6)$$

The terminology used here is obviously parallel to that used for beams. That is not the only parallel between beams and cubic splines. It is an interesting fact that a cubic spline is exactly the shape of a (linear) beam restrained to match the data by simple supports.

Note that the equations above are all linear equations with respect to the unknowns (coefficients). This feature makes splines easy to calculate since *solving linear systems is what computers do best*.

Exercises

19.1 Plot the following data, then try a polynomial fit of the correct degree to interpolate this number of data points:

```
> t = [ 0 .1 .499 .5 .6 1.0 1.4 1.5 1.899 1.9 2.0]
```

```
> y = [ 0 .06 .17 .19 .21 .26 .29 .29 .30 .31 .31]
```

What do you observe. Give an explanation of this error, in particular why is the term *badly conditioned* used?

19.2 Plot the following data along with a spline interpolant:

```
> t = [ 0 .1 .499 .5 .6 1.0 1.4 1.5 1.899 1.9 2.0]
```

```
> y = [ 0 .06 .17 .19 .21 .26 .29 .29 .30 .31 .31]
```

How does this compare with the plot above? What is a way to make the plot better?

Lecture 20

Least Squares Interpolation: Noisy Data

Very often data has a significant amount of noise. The least squares approximation is intentionally well-suited to represent noisy data. The next illustration shows the effects noise can have and how least squares is used.

Traffic flow model

Suppose you are interested in the time it takes to travel on a certain section of highway for the sake of planning. According to theory, assuming up to a moderate amount of traffic, the time should be approximately:

$$T(x) = ax + b$$

where b is the travel time when there is no other traffic, and x is the current number of cars on the road (in hundreds). To determine the coefficients a and b you could run several experiments which consist of driving the highway at different times of day and also estimating the number of cars on the road using a counter. Of course both of these measurements will contain *noise*, i.e. random fluctuations.

We could simulate such data in MATLAB as follows:

```
> x = 1:.1:6;  
> T = .1*x + 1;  
> Tn = T + .1*randn(size(x));  
> plot(x,Tn, 'r')
```

The data should look like it lies on a line, but with noise. Click on the **Tools** button and choose **Basic fitting**. Then choose a **linear** fit. The resulting line should go through the data in what looks like a very reasonable way. Click on **show equations**. Compare the equation with $T(x) = .1x + 1$. The coefficients should be pretty close considering the amount of noise in the plot. Next, try to fit the data with a spline. The result should be ugly. We can see from this example that **splines are not suited to noisy data**.

How does MATLAB obtain a very nice line to approximate noisy data? The answer is a very standard numerical/statistical method known as *least squares*.

Linear least squares

Consider in the previous example that we wish to fit a line to a lot of data that does not exactly lie on a line. For the equation of the line we have only two free coefficients, but we have many data points. We can not possibly make the line go through every data point, we can only wish for it to come reasonably close to as many data points as possible. Thus, our line must have an error with

respect to each data point. If $\ell(x)$ is our line and $\{(x_i, y_i)\}$ are the data, then

$$e_i = y_i - \ell(x_i)$$

is the error of ℓ with respect to each (x_i, y_i) . To make $\ell(x)$ reasonable, we wish to simultaneously minimize all the errors: $\{e_1, e_2, \dots, e_n\}$. There are many possible ways one could go about this, but the standard one is to try to minimize the *sum of the squares* of the errors. That is, we denote by \mathcal{E} the sum of the squares:

$$\mathcal{E} = \sum_{i=1}^n (y_i - \ell(x_i))^2 = \sum_{i=1}^n (y_i - ax_i - b)^2. \quad (20.1)$$

In the above expression x_i and y_i are given, but we are free to choose a and b , so we can think of \mathcal{E} as a function of a and b , i.e. $\mathcal{E}(a, b)$. In calculus, when one wishes to find a minimum value of a function of two variables, we set the partial derivatives equal to zero:

$$\begin{aligned} \frac{\partial \mathcal{E}}{\partial a} &= -2 \sum_{i=1}^n (y_i - ax_i - b) x_i = 0 \\ \frac{\partial \mathcal{E}}{\partial b} &= -2 \sum_{i=1}^n (y_i - ax_i - b) = 0. \end{aligned} \quad (20.2)$$

We can simplify these equations to obtain:

$$\begin{aligned} \left(\sum_{i=1}^n x_i^2 \right) a + \left(\sum_{i=1}^n x_i \right) b &= \sum_{i=1}^n x_i y_i \\ \left(\sum_{i=1}^n x_i \right) a + nb &= \sum_{i=1}^n y_i. \end{aligned} \quad (20.3)$$

Thus, the whole problem reduces to a 2 by 2 linear system to find the coefficients a and b . The entries in the matrix are determined from simple formulas using the data. The process is quick and easily automated, which is one reason it is very standard.

We could use the same process to obtain a quadratic or higher polynomial fit to data. If we try to fit an n degree polynomial, the software has to solve an $n \times n$ linear system, which is easily done. This is what MATLAB's basic fitting tool uses to obtain an n degree polynomial fit whenever the number of data points is more than $n + 1$.

Drag coefficients

Drag due to air resistance is proportional to the square of the velocity, i.e. $d = kv^2$. In a wind tunnel experiment the velocity v can be varied by setting the speed of the fan and the drag can be measured directly (it is the force on the object). In this and every experiment some random noise will occur. The following sequence of commands replicates the data one might receive from a wind tunnel:

```
> v = 0:1:60;
> d = .1234*v.^2;
> dn = d + .4*v.*randn(size(v));
> plot(v,dn,'*')
```

The plot should look like a quadratic, but with some noise. Using the tools menu, add a quadratic fit and enable the "show equations" option. What is the coefficient of x^2 ? How close is it to 0.1234?

Note that whenever you select a polynomial interpolation in MATLAB with a degree less than $n - 1$ MATLAB will produce a least squares interpolation.

You will notice that the quadratic fit includes both a constant and linear term. We know from the physical situation that these should not be there; they are remnants of noise and the fitting process. Since we know the curve should be kv^2 , we can do better by employing that knowledge. For instance, we know that the graph of d versus v^2 should be a straight line. We can produce this easily:

```
> vs = v.^2;
> plot(vs,dn,'*')
```

By changing the independent variable from v to v^2 we produced a plot that looks like a line with noise. Add a linear fit. What is the linear coefficient? This should be closer to 0.1234 than using a quadratic fit.

The second fit still has a constant term, which we know should not be there. If there was no noise, then at every data point we would have $k = d/v^2$. We can express this as a linear system $\mathbf{vs}'\mathbf{k} = \mathbf{dn}'$, which is badly overdetermined since there are more equations than unknowns. Since there is noise, each point will give a different estimate for k . In other words, the overdetermined linear system is also inconsistent. When MATLAB encounters such systems, it automatically gives a least squares solution of the matrix problem, i.e. one that minimizes the sum of the squared errors, which is exactly what we want. To get the least squares estimate for k , do

```
> k = vs'\dn'
```

This will produce a number close to .1234.

Note that this is an application where we have physical knowledge. In this situation extrapolation would be meaningful. For instance we could use the plot to find the predicted drag at 80 mph.

Exercises

20.1 Find two tables of data in an engineering textbook. Plot each and decide if the data is best represented by a polynomial, spline or least squares polynomial. Turn in the best plot of each. Indicate the source and meaning of the data.

20.2 The following table contains information from a chemistry experiment in which the concentration of a product was measured at one minute intervals.

T	0	1	2	3	4	5	6	7
C	3.033	3.306	3.672	3.929	4.123	4.282	4.399	4.527

Plot this data. Suppose it is known that this chemical reaction should follow the law: $c = a - b \exp(-0.2t)$. Following the example in the notes about the drag coefficients, change one of the variables so that the law is a linear function. Then plot the new variables and use the linear fit option to estimate a and b . What will be the eventual concentration? Finally, plot the graph of $a - b \exp(-0.2t)$ on the interval $[0,10]$, along with the data.

Lecture 21

Integration: Left, Right and Trapezoid Rules

The Left and Right endpoint rules

In this section, we wish to approximate a definite integral:

$$\int_a^b f(x) dx$$

where $f(x)$ is a continuous function. In calculus we learned that integrals are (signed) areas and can be approximated by sums of smaller areas, such as the areas of rectangles. We begin by choosing points $\{x_i\}$ that subdivide $[a, b]$:

$$a = x_0 < x_1 < \dots < x_{n-1} < x_n = b.$$

The subintervals $[x_{i-1}, x_i]$ determine the width Δ_i of each of the approximating rectangles. For the height, we learned that we can choose any height of the function $f(x_i^*)$ where $x_i^* \in [x_{i-1}, x_i]$. The resulting approximation is:

$$\int_a^b f(x) dx \approx \sum_{i=1}^n f(x_i^*) \Delta_i.$$

To use this to approximate integrals with actual numbers, we need to have a specific x_i^* in each interval. The two simplest (and worst) ways to choose x_i^* are as the left-hand point or the right-hand point of each interval. This gives concrete approximations which we denote by L_n and R_n given by

$$L_n = \sum_{i=1}^n f(x_{i-1}) \Delta_i \quad \text{and}$$
$$R_n = \sum_{i=1}^n f(x_i) \Delta_i.$$

```
function L = myleftsum(x,y)
% produces the left sum from data input.
% Inputs: x -- vector of the x coordinates of the partition
%         y -- vector of the corresponding y coordinates
% Output: returns the approximate integral
n = max(size(x)); % safe for column or row vectors
L = 0;
for i = 1:n-1
    L = L + y(i)*(x(i+1) - x(i));
end
```

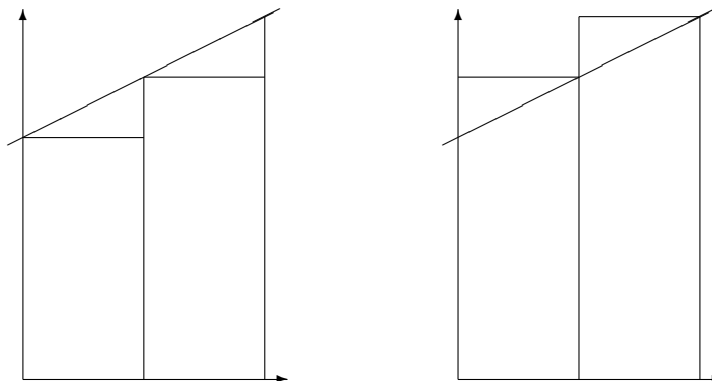


Figure 21.1: The left and right sums, L_n and R_n .

Often we can take $\{x_i\}$ to be *evenly spaced*, with each interval having the same width:

$$h = \frac{b-a}{n},$$

where n is the number of subintervals. If this is the case, then L_n and R_n simplify to:

$$L_n = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i) \quad \text{and} \quad (21.1)$$

$$R_n = \frac{b-a}{n} \sum_{i=1}^n f(x_i). \quad (21.2)$$

The foolishness of choosing left or right endpoints is illustrated in Figure 21.1. As you can see, for a very simple function like $f(x) = 1 + .5x$, each rectangle of L_n is too short, while each rectangle of R_n is too tall. This will hold for any increasing function. For decreasing functions L_n will always be too large while R_n will always be too small.

The Trapezoid rule

Knowing that the errors of L_n and R_n are of opposite sign, a very reasonable way to get a better approximation is to take an average of the two. We will call the new approximation T_n :

$$T_n = \frac{L_n + R_n}{2}.$$

This method also has a straight-forward geometric interpretation. On each subrectangle we are using:

$$A_i = \frac{f(x_{i-1}) + f(x_i)}{2} \Delta_i,$$

which is exactly the area of the *trapezoid* with sides $f(x_{i-1})$ and $f(x_i)$. We thus call the method the trapezoid method. See Figure 21.2.

We can rewrite T_n as

$$T_n = \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} \Delta_i.$$

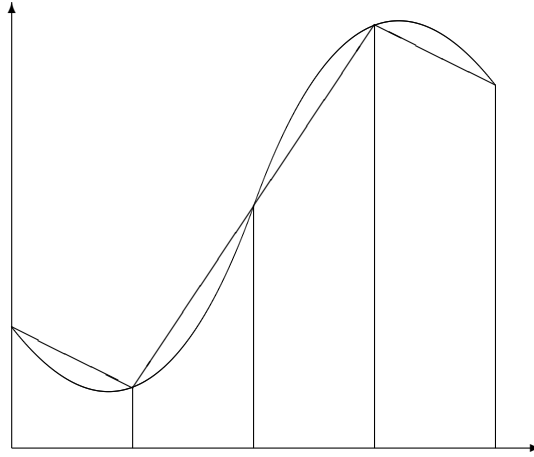


Figure 21.2: The trapezoid rule, T_n .

In the evenly spaced case, we can write this as

$$T_n = \frac{b-a}{2n} (f(x_0) + 2f(x_1) + \dots + 2f(x_{n-1}) + f(x_n)). \quad (21.3)$$

Caution: The convention used here is to begin numbering the points at 0, i.e. $x_0 = a$; this allows n to be the number of subintervals and the index of the last point x_n . However, MATLAB's indexing convention begins at 1. Thus, when programming in MATLAB, the first entry in \mathbf{x} will be x_0 , i.e. $\mathbf{x}(1) = x_0$ and $\mathbf{x}(n+1) = x_n$.

If we are given data about the function, rather than a formula for the function, often the data are not evenly spaced. The following function program could then be used.

```
function T = mytrap(x,y)
% Calculates the Trapezoid rule approximation of the integral from input data
% Inputs: x -- vector of the x coordinates of the partition
%         y -- vector of the corresponding y coordinates
% Output: returns the approximate integral
n = max(size(x)); % safe for column or row vectors
T = 0;
for i = 1:n-1
    T = T + .5*(y(i)+y(i+1))*(x(i+1) - x(i));
end
```

Using the Trapezoid rule for areas in the plane

In multi-variable calculus you were supposed to learn that you can calculate the area of a region R in the plane by calculating the line integral:

$$A = - \oint_C y dx, \quad (21.4)$$

where C is the counter-clockwise curve around the boundary of the region. We can represent such a curve by consecutive points on it, i.e. $\bar{x} = (x_0, x_1, x_2, \dots, x_{n-1}, x_n)$, and $\bar{y} = (y_0, y_1, y_2, \dots, y_{n-1}, y_n)$.

Since we are assuming the curve ends where it begins, we require $(x_n, y_n) = (x_0, y_0)$. Applying the trapezoid method to the integral (21.4) gives

$$A = - \sum_{i=1}^n \frac{y_{i-1} + y_i}{2} (x_i - x_{i-1}).$$

This formula then is the basis for calculating areas when coordinates of boundary points are known, but not necessarily formulas for the boundaries such as in a land survey.

In the following script, we can use this method to approximate the area of a unit circle using n points on the circle:

```
% Calculates pi using a trapezoid approximation of the unit circle.
format long
n = 10;
t = linspace(0,2*pi,n+1);
x = cos(t);
y = sin(t);
plot(x,y)
A = 0
for i = 1:n
    A = A - (y(i)+y(i+1))*(x(i+1)-x(i))/2;
end
A
```

Exercises

21.1 For the integral $\int_1^2 \sqrt{x} dx$ calculate L_4 , R_4 , and T_4 with even spacing (by hand, but use a calculator) using formulas (21.1), (21.2) and (21.3). Find the percentage error of these approximations, using the exact value.

21.2 Write a function program `myints` whose inputs are f , a , b and n and whose outputs are L , R and T , the left, right and trapezoid integral approximations for f on $[a, b]$ with n subintervals. To make it efficient, first use `x = linspace(a,b,n+1)` to make the x values and `y = f(x)` to make the y values, then add the 2nd to n th y entries only once (use only one loop in the program).

Change to `format long` and apply your program to the integral $\int_1^2 \sqrt{x} dx$. Compare with the results of the previous exercise. Also find L_{100} , R_{100} and T_{100} and the relative errors of these approximations.

Turn in the program and a brief summary of the results.

Lecture 22

Integration: Midpoint and Simpson's Rules

Midpoint rule

If we use the endpoints of the subintervals to approximate the integral, we run the risk that the values at the endpoints do not accurately represent the average value of the function on the subinterval. A point which is much more likely to be close to the average would be the midpoint of each subinterval. Using the midpoint in the sum is called the *midpoint rule*. On the i -th interval $[x_{i-1}, x_i]$ we will call the midpoint \bar{x}_i , i.e.

$$\bar{x}_i = \frac{x_{i-1} + x_i}{2}.$$

If $\Delta_i = x_i - x_{i-1}$ is the length of each interval, then using midpoints to approximate the integral would give the formula:

$$M_n = \sum_{i=1}^n f(\bar{x}_i) \Delta_i.$$

For even spacing, $\Delta = (b - a)/n$, and the formula is:

$$M_n = \frac{b - a}{n} \sum_{i=1}^n f(\bar{x}_i) = \frac{b - a}{n} (\hat{y}_1 + \hat{y}_2 + \dots + \hat{y}_n), \quad (22.1)$$

where we define $\hat{y}_i = f(\bar{x}_i)$.

While the midpoint method is obviously better than L_n or R_n , it is not obvious that it is actually better than the trapezoid method T_n , but it is.

Simpson's rule

Consider Figure 22.1. If f is not linear on a subinterval, then it can be seen that the errors for the midpoint and trapezoid rules behave in a very predictable way, they have opposite sign. For example, if the function is concave up then T_n will be too high, while M_n will be too low. Thus it makes sense that a better estimate would be to average T_n and M_n . However, in this case we can do better than a simple average. The error will be minimized if we use a weighted average. To find the proper weight we take advantage of the fact that for a quadratic function the errors EM_n and ET_n are exactly related by:

$$|EM_n| = \frac{1}{2}|ET_n|.$$

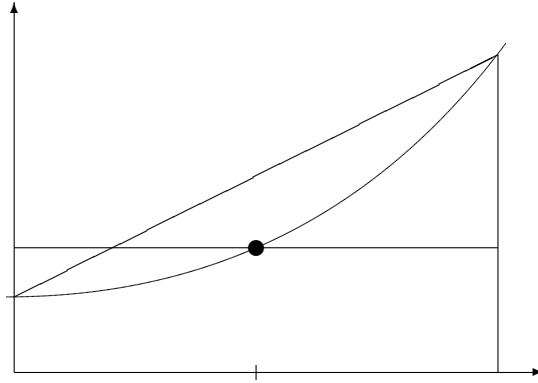


Figure 22.1: Comparing the trapezoid and midpoint method on a single subinterval. The function is concave up, in which case T_n is too high, while M_n is too low.

Thus we take the following weighted average of the two, which is called Simpson's rule:

$$S_{2n} = \frac{2}{3}M_n + \frac{1}{3}T_n.$$

If we use this weighting on a quadratic function the two errors will exactly cancel.

Notice that we write the subscript as $2n$. That is because we usually think of $2n$ subintervals in the approximation; the n subintervals of the trapezoid are further subdivided by the midpoints. We can then number all the points using integers. If we number them this way we notice that the number of subintervals must be an even number.

The formula for Simpson's rule if the subintervals are evenly spaced is the following (with n intervals, where n is even):

$$S_n = \frac{b-a}{3n} (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)).$$

Note that if we are presented with data $\{x_i, y_i\}$ where the x_i points are evenly spaced with $x_{i+1} - x_i = \Delta x$, it is easy to apply Simpson's method:

$$S_n = \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + \dots + 2y_{n-2} + 4y_{n-1} + y_n). \quad (22.2)$$

Notice the pattern of the coefficients. The following program will produce these coefficients for n intervals, if n is an even number. Try it for $n = 6, 7, 100$.

```

function w = mysimpweights(n)
% computes the weights for Simpson's rule
% Input: n -- the number of intervals, must be even
% Output: a vector with the weights, length n+1
if rem(n,2) ~= 0
    error('n must be even')
end
w = ones(n+1,1);
for i = 2:n
    if rem(i,2)==0
        w(i)=4;
    else
        w(i)=2;
    end
end
end

```

Simpson's rule is incredibly accurate. We will consider just how accurate in the next section. The one drawback is that the points used must either be evenly spaced, or at least the odd number points must lie exactly at the midpoint between the even numbered points. In applications where you can choose the spacing, this is not a problem. In applications such as experimental or simulated data, you might not have control over the spacing and then you cannot use Simpson's rule.

Error bounds

The trapezoid, midpoint, and Simpson's rules are all approximations. As with any approximation, before you can safely use it, you must know how good (or bad) the approximation might be. For these methods there are formulas that give upper bounds on the error. In other words, the worst case errors for the methods. These bounds are given by the following statements:

- Suppose f'' is continuous on $[a,b]$. Let $K_2 = \max_{x \in [a,b]} |f''(x)|$. Then the errors ET_n and EM_n of the Trapezoid and Midpoint rules, respectively, applied to $\int_a^b f dx$ satisfy:

$$|ET_n| \leq \frac{K_2(b-a)^3}{12n^2} = \frac{K_2(b-a)}{12} \Delta x^2 \quad \text{and}$$

$$|EM_n| \leq \frac{K_2(b-a)^3}{24n^2} = \frac{K_2(b-a)}{24} \Delta x^2.$$

- Suppose $f^{(4)}$ is continuous on $[a,b]$. Let $K_4 = \max_{x \in [a,b]} |f^{(4)}(x)|$. Then the error ES_n of Simpson's rule applied to $\int_a^b f dx$ satisfies:

$$|ES_n| \leq \frac{K_4(b-a)^5}{180n^4} = \frac{K_4(b-a)}{180} \Delta x^4.$$

In practice K_2 and K_4 are themselves approximated from the values of f at the evaluation points.

The most important thing in these error bounds is the last term. For the trapezoid and midpoint method, the error depends on Δx^2 whereas the error for Simpson's rule depends on Δx^4 . If Δx is just moderately small, then there is a huge advantage with Simpson's method.

When an error depends on a power of a parameter, such as above, we sometimes use the **order notation**. In the above error bounds we say that the trapezoid and midpoint rules have errors of order $O(\Delta x^2)$, whereas Simpson's rule has error of order $O(\Delta x^4)$.

In MATLAB there is a built-in command for definite integrals: `quad(f,a,b)` where the `f` is an inline function and `a` and `b` are the endpoints. Here `quad` stands for quadrature, which is a term for numerical integration. The command uses “adaptive Simpson quadrature”, a form of Simpson’s rule that checks its own accuracy and adjusts the grid size where needed. Here is an example of its usage:

```
> f = inline('x.^(1/3).*sin(x.^3)')
> I = quad(f,0,1)
```

Exercises

- 22.1 Using formulas (22.1) and (22.2), for the integral $\int_1^2 \sqrt{x} dx$ calculate M_4 and S_4 (by hand, but use a calculator). Find the percentage error of these approximations, using the exact value. Compare with exercise 21.1.
- 22.2 Write a function program `mymidpoint` that calculates the midpoint rule approximation for $\int f$ on the interval $[a, b]$ with n subintervals. The inputs should be f , a , b and n . Use your program on the integral $\int_1^2 \sqrt{x} dx$ to obtain M_4 and M_{100} . Compare these with the previous exercise and the true value of the integral.
- 22.3 Write a function program `mysimpson` that calculates the Simpson’s rule approximation for $\int f$ on the interval $[a, b]$ with n subintervals. It should use the program `mysimpweights` to produce the coefficients. Use your program on the integral $\int_1^2 \sqrt{x} dx$ to obtain S_4 and S_{100} . Compare these with the previous exercise and the true value.

Lecture 23

Plotting Functions of Two Variables

Functions on Rectangular Grids

Suppose you wish to plot a function $f(x, y)$ on the rectangle $a \leq x \leq b$ and $c \leq y \leq d$. The graph of a function of two variables is of course a three dimensional object. Visualizing the graph is often very useful.

For example, suppose you have a formula:

$$f(x, y) = x \sin(xy)$$

and you are interested in the function on the region $0 \leq x \leq 5$, $\pi \leq y \leq 2\pi$. A way to plot this function in MATLAB would be the following sequence of commands:

```
> f = inline('x.*sin(x.*y)', 'x', 'y')
> [X,Y] = meshgrid(0:.1:5,pi:.01*pi:2*pi);
> Z = f(X,Y)
> mesh(X,Y,Z)
```

This will produce a 3-D plot that you can rotate by clicking on the rotate icon and then dragging with the mouse.

Instead of the command `mesh`, you could use the command:

```
> surf(X,Y,Z)
```

The key command in this sequence is `[X Y] = meshgrid(a:h:b,c:k:d)`, which produces *matrices* of x and y values in X and Y . Enter:

```
> size(X)
> size(Y)
> size(Z)
```

to see that each of these variables is a 101×51 matrix. To see the first few entries of X enter:

```
> X(1:6,1:6)
```

and to see the first few values of Y type:

```
> Y(1:6,1:6)
```

You should observe that the x values in X begin at 0 on the left column and increase from left to right. The y values on the other have start at π at the top and increase from top to bottom. Note that this arrangement is flipped from the usual arrangement in the x - y plane.

In the command `[X Y] = meshgrid(a:h:b,c:k:d)`, h is the increment in the x direction and k is the increment in the y direction. Often we will calculate:

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

where m is the number of *intervals* in the x direction and n is the number of intervals in the y direction. To obtain a good plot it is best if m and n can be set between 10 and 100.

For another example of how `meshgrid` works, try the following and look at the output in `X` and `Y`.

```
> [X,Y] = meshgrid(0:.5:4,1:.2:2);
```

Scattered Data and Triangulation

Often we are interested in objects whose bases are not rectangular. For instance, data does not usually come arranged in a nice rectangular grid; rather, measurements are taken where convenient.

In MATLAB we can produce triangles for a region by recording the coordinates of the vertices and recording which vertices belong to each triangle. The following script program produces such a set of triangles:

```
% mytriangles
% Program to produce a triangulation.
% V contains vertices, which are (x,y) pairs
V = [ 1/2 1/2 ; 1 1 ; 3/2 1/2 ; .5 1.5 ; 0 0
      1 0 ; 2 0 ; 2 1 ; 1.5 1.5 ; 1 2
      0 2 ; 0 1]
% x, y are row vectors containing coordinates of vertices
x = V(:,1)';
y = V(:,2)';
% Assign the triangles
T = delaunay(x,y)
```

You can plot the triangles using the following command:

```
> trimesh(T,x,y)
```

You can also prescribe values (heights) at each vertex directly (say from a survey):

```
> z1 = [ 2 3 2.5 2 1 1 .5 1.5 1.6 1.7 .9 .5 ];
```

or using a function:

```
> f = inline('abs(sin(x.*y)).^(3/2)', 'x', 'y');
```

```
> z2 = f(x,y);
```

The resulting profiles can be plotted:

```
> trimesh(T,x,y,z1)
```

```
> trisurf(T,x,y,z2)
```

Each row of the matrix `T` corresponds to a triangle, so `T(i,:)` gives triangle number `i`. The three corners of triangle number `i` are at indices `T(i,1)`, `T(i,2)`, and `T(i,3)`. So for example to get the y -coordinate of the second point of triangle number 5, enter:

```
> y(T(5,2))
```

To see other examples of regions defined by triangle get `mywedge.m` and `mywasher.m` from the class web site and run them. Each of these programs defines vectors `x` and `y` of x and y values of vertices and a matrix `T`. As before `T` is a list of sets of three integers. Each triple of integers indicates which vertices to connect in a triangle.

To plot a function, say $f(x,y) = x^2 - y^2$ on the washer figure try:

```
> mywasher
```

```
> z = x.^2 - y.^2
```

```
> trisurf(T,x,y,z)
```

Note again that this plot can be rotated using the icon and mouse.

Exercises

- 23.1 Plot the function $f(x, y) = xe^{-x^2-y^2}$ on the rectangle $-3 \leq x \leq 3$, $-2 \leq y \leq 2$ using meshgrid. Make an appropriate choice of h and k and if necessary a rotation to produce a good plot. Turn in your plot and the calculation of k and h .
- 23.2 Make up an interesting or useful convex irregular region and choose well-distributed points in it and on the boundary as a basis for a triangulation. Make sure to include a good number of both interior and boundary points. (Graph paper would be handy.) Record the coordinates of these points in the matrix V in a copy of the program `mytriangles.m`. Plot the triangulation. Also create values at the vertices using a function and plot the resulting profile.

Lecture 24

Double Integrals for Rectangles

The center point method

Suppose that we need to find the integral of a function, $f(x, y)$, on a rectangle:

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

In calculus you learned to do this by an iterated integral:

$$I = \iint_R f \, dA = \int_a^b \int_c^d f(x, y) \, dy \, dx = \int_c^d \int_a^b f(x, y) \, dx \, dy.$$

You also should have learned that the integral is the limit of the Riemann sums of the function as the size of the subrectangles goes to zero. This means that the Riemann sums are approximations of the integral, which is precisely what we need for numerical methods.

For a rectangle R , we begin by subdividing into smaller subrectangles $\{R_{ij}\}$, in a systematic way. We will divide $[a, b]$ into m subintervals and $[c, d]$ into n subintervals. Then R_{ij} will be the “intersection” of the i -th subinterval in $[a, b]$ with the j -th subinterval of $[c, d]$. In this way the entire rectangle is subdivided into mn subrectangles, numbered as in Figure 24.1.

A Riemann sum using this subdivision would have the form:

$$S = \sum_{i,j=1,1}^{m,n} f(x_{ij}^*) A_{ij} = \sum_{j=1}^n \left(\sum_{i=1}^m f(x_{ij}^*) A_{ij} \right),$$

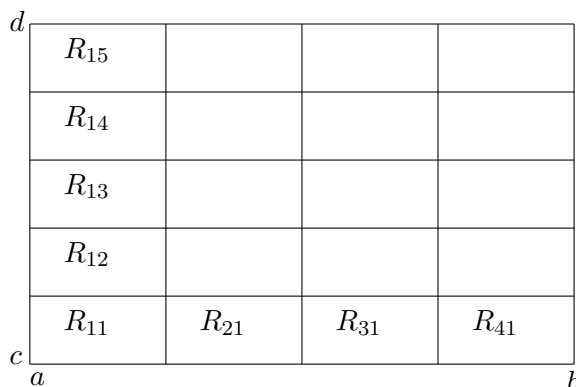


Figure 24.1: Subdivision of the rectangle $R = [a, b] \times [c, d]$ into subrectangles R_{ij} .

where $A_{ij} = \Delta x_i \Delta y_j$ is the area of R_{ij} , and x_{ij}^* is a point in R_{ij} . The theory of integrals tells us that if f is continuous, then this sum will converge to the same number, no matter how we choose x_{ij}^* . For instance, we could choose x_{ij}^* to be the point in the lower left corner of R_{ij} and the sum would still converge as the size of the subrectangles goes to zero. However, in practice we wish to choose x_{ij}^* in such a way to make S as accurate as possible even when the subrectangles are not very small. The obvious choice for the best point in R_{ij} would be the center point. The center point is most likely of all points to have a value of f close to the average value of f . If we denote the center points by c_{ij} , then the sum becomes

$$C_{mn} = \sum_{i,j=1,1}^{m,n} f(c_{ij}) A_{ij}.$$

Here

$$c_{ij} = \left(\frac{x_{i-1} + x_i}{2}, \frac{y_{j-1} + y_j}{2} \right).$$

Note that if the subdivision is evenly spaced then $\Delta x \equiv (b - a)/m$ and $\Delta y \equiv (d - c)/n$, and so in that case

$$C_{mn} = \frac{(b - a)(d - c)}{mn} \sum_{i,j=1,1}^{m,n} f(c_{ij}).$$

The four corners method

Another good idea would be to take the value of f not only at one point, but as the average of the values at several points. An obvious choice would be to evaluate f at all four corners of each R_{ij} then average those. If we note that the lower left corner is (x_i, y_j) , the upper left is (x_i, y_{j+1}) , the lower right is (x_{i+1}, y_j) and the upper right is (x_{i+1}, y_{j+1}) , then the corresponding sum will be

$$F_{mn} = \sum_{i,j=1,1}^{m,n} \frac{1}{4} (f(x_i, y_j) + f(x_i, y_{j+1}) + f(x_{i+1}, y_j) + f(x_{i+1}, y_{j+1})) A_{ij},$$

which we will call the *four-corners* method. If the subrectangles are evenly spaced, then we can simplify this expression. Notice that $f(x_i, y_j)$ gets counted multiple times depending on where (x_i, y_j) is located. For instance if (x_i, y_j) is in the interior of R then it is the corner of 4 subrectangles. Thus the sum becomes

$$F_{mn} = \frac{A}{4} \left(\sum_{\text{corners}} f(x_i, y_j) + 2 \sum_{\text{edges}} f(x_i, y_j) + 4 \sum_{\text{interior}} f(x_i, y_j) \right),$$

where $A = \Delta x \Delta y$ is the area of the subrectangles. We can think of this as a weighted average of the values of f at the grid points (x_i, y_j) . The weights used are represented in the matrix

$$W = \begin{pmatrix} 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \end{pmatrix}. \quad (24.1)$$

We could implement the four-corner method by forming a matrix (f_{ij}) of f values at the grid points, then doing entry-wise multiplication of the matrix with the weight matrix. Then the integral would

be obtained by summing all the entries of the resulting matrix and multiplying that by $A/4$. The formula would be

$$F_{mn} = \frac{(b-a)(d-c)}{4mn} \sum_{i,j=1,1}^{m+1,n+1} W_{ij} f(x_i, y_j).$$

Notice that the four-corners method coincides with applying the trapezoid rule in each direction. Thus it is in fact a *double trapezoid* rule.

The double Simpson method

The next improvement one might make would be to take an average of the center point sum C_{mn} and the four corners sum F_{mn} . However, a more standard way to obtain a more accurate method is the Simpson double integral. It is obtained by applying Simpson's rule for single integrals to the iterated double integral. The resulting method requires that both m and n be even numbers and the grid be evenly spaced. If this is the case we sum up the values $f(x_i, y_j)$ with weights represented in the matrix

$$W = \begin{pmatrix} 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \end{pmatrix}. \quad (24.2)$$

The sum of the weighted values is multiplied by $A/9$ and the formula is

$$S_{mn} = \frac{(b-a)(d-c)}{9mn} \sum_{i,j=1,1}^{m+1,n+1} W_{ij} f(x_i, y_j).$$

MATLAB has a built in command for double integrals on rectangles: `dblquad(f,a,b,c,d)`. Here is an example:

```
> f = inline('sin(x.*y)./sqrt(x+y)', 'x', 'y')
> I = dblquad(f,0.5,1,0.5,2)
```

Below is a MATLAB function which will produce the matrix of weights needed for Simpson's rule for double integrals. It uses the function `mysimpweights` from Lecture 22.

```
function W = mydblsimpweights(m,n)
% Produces the m by n matrix of weights for Simpson's rule
% for double integrals
% Inputs: m -- number of intervals in the row direction.
%          must be even.
%          n -- number of intervals in the column direction.
%          must be even.
% Output: W -- a (m+1)x(n+1) matrix of the weights
if rem(m,2)~=0 | rem(n,2)~=0
    error('m and n must be even')
end
u = mysimpweights(m);
v = mysimpweights(n);
W = u*v';
```

Exercises

- 24.1 Download the program `mylowerleft` from the web site. Modify it to make a program `mycenter` that does the center point method. Implement the change by changing the “mesh-grid” to put the grid points at the centers of the subrectangles. Look at the mesh plot produced to make sure the program is putting the grid where you want it. Use both programs to approximate the integral

$$\int_0^2 \int_1^5 \sqrt{xy} \, dy \, dx,$$

using $(m, n) = (10, 18)$. Evaluate this integral exactly to make comparisons.

- 24.2 Write a program `mydblsimp` that computes the Simpson’s rule approximation. Let it use the program `mydblsimpweights` to make the weight matrix (24.2). Check the accuracy of the program on the integral in the previous problem.
- 24.3 Using `mysimpweights` and `mydblsimpweights` as models make programs `mytrapweights` and `mydbltrapweights` that will produce the weights for the trapezoid rule and the weight matrix for the four corners (double trapezoid) method (24.1).

Lecture 25

Double Integrals for Non-rectangles

In the previous lecture we considered only integrals over rectangular regions. In practice, regions of interest are rarely rectangles and so in this lecture we consider two strategies for evaluating integrals over other regions.

Redefining the function

One strategy is to redefine the function so that it is zero outside the region of interest, then integrate over a rectangle that includes the region.

For example, suppose we need to approximate the value of

$$I = \iint_T \sin^3(xy) \, dx \, dy$$

where T is the triangle with corners at $(0, 0)$, $(1, 0)$ and $(0, 2)$. Then we could let R be the rectangle $[0, 1] \times [0, 2]$ which contains the triangle T . Notice that the hypotenuse of the triangle has the equation $2x + y = 2$. Then make $f(x) = \sin^3(xy)$ if $2x + y \leq 2$ and $f(x) = 0$ if $2x + y > 2$. In MATLAB we can make this function with the command:

```
> f = inline('sin(x.*y).^3.*(2*x + y <= 2)')
```

In this command `<=` is a *logical* command. The term in parentheses is then a *logical statement* and is given the value 1 if the statement is true and 0 if it is false. We can then integrate the modified f on $[0, 1] \times [0, 2]$ using the command:

```
> I = dblquad(f,0,1,0,2)
```

As another example, suppose we need to integrate $x^2 \exp(xy)$ inside the circle of radius 2 centered at $(1, 2)$. The equation for this circle is $(x - 1)^2 + (y - 2)^2 = 4$. Note that the inside of the circle is $(x - 1)^2 + (y - 2)^2 \leq 4$ and that the circle is contained in the rectangle $[-1, 3] \times [0, 4]$. Thus we can create the right function and integrate it by:

```
> f = inline('x.^2.*exp(x.*y).*((x-1).^2 + (y-2).^2 <= 4)')
```

```
> I = dblquad(f,-1,3,0,4)
```

Integration Based on Triangles

The second approach to integrating over non-rectangular regions, is based on subdividing the region into triangles. Such a subdivision is called a *triangulation*. On regions where the boundary consists of line segments, this can be done exactly. Even on regions where the boundary contains curves, this can be done approximately. This is a very important idea for several reasons, the most important of which is that the finite elements method is based on it. Another reason this is important is that often the values of f are not given by a formula, but from data. For example, suppose you are surveying on a construction site and you want to know how much fill will be needed to bring the level up to the plan. You would proceed by taking elevations at numerous points across the site. However, if the site is irregularly shaped or if there are obstacles on the site, then you cannot make these measurements on an exact rectangular grid. In this case, you can use triangles by connecting your points with triangles. Many software packages will even choose the triangles for you (MATLAB will do it using the command `delaunay`).

The basic idea of integrals based on triangles is exactly the same as that for rectangles; the integral is approximated by a sum where each term is a value times an area

$$I \approx \sum_{i=1}^n f(x_i^*) A_i,$$

where n is the number of triangles, A_i is the area of the triangle and x^* a point in the triangle. However, rather than considering the value of f at just one point people often consider an average of values at several points. The most convenient of these is of course the corner points. We can represent this sum by

$$T_n = \sum_{i=1}^n \bar{f}_i A_i,$$

where \bar{f} is the average of f at the corners.

If the triangle has vertices (x_1, y_1) , (x_2, y_2) and (x_3, y_3) , the formula for area is

$$A = \frac{1}{2} \left| \det \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \right|. \quad (25.1)$$

A function `mythreecorners` to compute using the three corners method is given below.

Another idea would be to use the center point (centroid) of each triangle. If the triangle has vertices (x_1, y_1) , (x_2, y_2) and (x_3, y_3) , then the centroid is given by the simple formulas

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3} \quad \text{and} \quad \bar{y} = \frac{y_1 + y_2 + y_3}{3}. \quad (25.2)$$

```

function I = mythreecorners(f,V,T)
% Integrates a function based on a triangulation, using three corners
% Inputs: f -- the function to integrate, as an inline
%         V -- the vertices. Each row has the x and y coordinates of a vertex
%         T -- the triangulation. Each row gives the indices of three corners
% Output: the approximate integral

x = V(:,1); % extract x and y coordinates of all nodes
y = V(:,2);
I=0;
p = size(T,1);
for i = 1:p
    x1 = x(T(i,1)); % find coordinates and area
    x2 = x(T(i,2));
    x3 = x(T(i,3));
    y1 = y(T(i,1));
    y2 = y(T(i,2));
    y3 = y(T(i,3));
    A = .5*abs(det([x1, x2, x3; y1, y2, y3; 1, 1, 1]));
    z1 = f(x1,y1); % find values and average
    z2 = f(x2,y2);
    z3 = f(x3,y3);
    zavg = (z1 + z2 + z3)/3;
    I = I + zavg*A; % accumulate integral
end

```

Exercises

- 25.1 Download the programs `mywedge.m` and `mywasher.m` from the web site. Plot $f(x, y) = \frac{x+y}{x^2+y^2}$ on the region produced by `mywasher.m` and $f(x, y) = \sin(x) + \sqrt{y}$ on the region produced by `mywedge.m`.
- 25.2 Modify the program `mythreecorners.m` to a new program `mycenters.m` that does the centerpoint method for triangles. Run the program on the region produced by `mywasher.m` with the function $f(x, y) = \frac{x+y}{x^2+y^2}$ and on the region produced by `mywedge.m` with the function $f(x, y) = \sin(x) + \sqrt{y}$.

Lecture 26

Gaussian Quadrature*

Exercises

26.1

26.2

Lecture 27

Numerical Differentiation

Approximating derivatives from data

Suppose that a variable y depends on another variable x , i.e. $y = f(x)$, but we only know the values of f at a finite set of points, e.g., as data from an experiment or a simulation:

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n).$$

Suppose then that we need information about the derivative of $f(x)$. One obvious idea would be to approximate $f'(x_i)$ by the **Forward Difference**:

$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

This formula follows directly from the definition of the derivative in calculus. An alternative would be to use a **Backward Difference**:

$$f'(x_i) \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}.$$

Since the errors for the forward difference and backward difference tend to have opposite signs, it would seem likely that averaging the two methods would give a better result than either alone. If the points are evenly spaced, i.e. $x_{i+1} - x_i = x_i - x_{i-1} = h$, then averaging the forward and backward differences leads to a symmetric expression called the **Central Difference**:

$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_{i-1}}{2h}.$$

Errors of approximation

We can use Taylor polynomials to derive the accuracy of the forward, backward and central difference formulas. For example the usual form of the Taylor polynomial with remainder (sometimes called Taylor's Theorem) is:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(c),$$

where c is some (unknown) number between x and $x+h$. Letting $x = x_i$, $x+h = x_{i+1}$ and solving for $f'(x_i)$ leads to:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{h}{2}f''(c).$$

Notice that the quotient in this equation is exactly the forward difference formula. Thus the error of the forward difference is $-(h/2)f''(c)$ which means it is $O(h)$. Replacing h in the above calculation

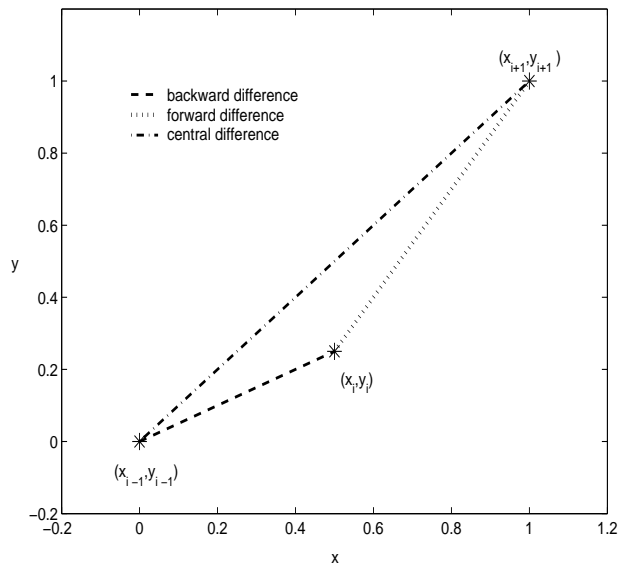


Figure 27.1: The three difference approximations of y'_i .

by $-h$ gives the error for the backward difference formula; it is also $O(h)$. For the central difference, the error can be found from the third degree Taylor polynomial with remainder:

$$f(x_{i+1}) = f(x_i + h) = f(x_i) + hf'(x_i) + \frac{h^2}{2}f''(x_i) + \frac{h^3}{3!}f'''(c_1) \quad \text{and}$$

$$f(x_{i-1}) = f(x_i - h) = f(x_i) - hf'(x_i) + \frac{h^2}{2}f''(x_i) - \frac{h^3}{3!}f'''(c_2)$$

where $x_i \leq c_1 \leq x_{i+1}$ and $x_{i-1} \leq c_2 \leq x_i$. Subtracting these two equations and solving for $f'(x_i)$ leads to:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} - \frac{h^2}{3!} \frac{f'''(c_1) + f'''(c_2)}{2}.$$

This shows that the error for the central difference formula is $O(h^2)$. Thus, central differences are significantly better and so: **It is best to use central differences whenever possible.**

There are also central difference formulas for higher order derivatives. These all have error of order $O(h^2)$:

$$f''(x_i) = y_i'' \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2},$$

$$f'''(x_i) = y_i''' \approx \frac{1}{2h^3} [y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}], \quad \text{and}$$

$$f^{(4)}(x_i) = y_i^{(4)} \approx \frac{1}{h^4} [y_{i+2} - 4y_{i+1} + 6y_i - 4y_{i-1} + y_{i-2}].$$

Partial Derivatives

Suppose $u = u(x, y)$ is a function of two variables that we only know at grid points (x_i, y_j) . We will use the notation:

$$u_{i,j} = u(x_i, y_j)$$

frequently throughout the rest of the lectures. We can suppose that the grid points are evenly spaced, with an increment of h in the x direction and k in the y direction. The central difference formulas for the partial derivatives would be:

$$u_x(x_i, y_j) \approx \frac{1}{2h} (u_{i+1,j} - u_{i-1,j}) \quad \text{and}$$

$$u_y(x_i, y_j) \approx \frac{1}{2k} (u_{i,j+1} - u_{i,j-1}).$$

The second partial derivatives are:

$$u_{xx}(x_i, y_j) \approx \frac{1}{h^2} (u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) \quad \text{and}$$

$$u_{yy}(x_i, y_j) \approx \frac{1}{k^2} (u_{i,j+1} - 2u_{i,j} + u_{i,j-1}),$$

and the mixed partial derivative is:

$$u_{xy}(x_i, y_j) \approx \frac{1}{4hk} (u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}).$$

Caution: Notice that we have indexed u_{ij} so that as a matrix each row represents the values of u at a certain x_i and each column contains values at y_j . The arrangement in the matrix does not coincide with the usual orientation of the xy -plane.

Let's consider an example. Let the values of u at (x_i, y_j) be recorded in the matrix:

$$(u_{ij}) = \begin{pmatrix} 5.1 & 6.5 & 7.5 & 8.1 & 8.4 \\ 5.5 & 6.8 & 7.8 & 8.3 & 8.9 \\ 5.5 & 6.9 & 9.0 & 8.4 & 9.1 \\ 5.4 & 9.6 & 9.1 & 8.6 & 9.4 \end{pmatrix} \quad (27.1)$$

Assume the indices begin at 1, i is the index for rows and j the index for columns. Suppose that $h = .5$ and $k = .2$. Then $u_y(x_2, y_4)$ would be approximated by the central difference:

$$u_y(x_2, y_4) \approx \frac{u_{2,5} - u_{2,3}}{2k} \approx \frac{8.9 - 7.8}{2 \cdot 0.2} = 2.75.$$

The partial derivative $u_{xy}(x_2, y_4)$ is approximated by:

$$u_{xy}(x_2, y_4) \approx \frac{u_{3,5} - u_{3,3} - u_{1,5} + u_{1,3}}{4hk} \approx \frac{9.1 - 9.0 - 8.4 + 7.5}{4 \cdot .5 \cdot .2} = -2.$$

Exercises

27.1 Suppose you are given the data in the following table.

t	0	.5	1.0	1.5	2.0
y	0	.19	.26	.29	.31

- Give the forward, backward and central difference approximations of $f'(1)$.
- Give the central difference approximations for $f''(1)$, $f'''(1)$ and $f^{(4)}(1)$.

27.2 Suppose values of $u(x, y)$ at points (x_i, y_j) are given in the matrix (27.1). Suppose that $h = .1$ and $k = .5$. Approximate the following derivatives by central differences:

- $u_x(x_2, y_4)$
- $u_{xx}(x_3, y_2)$
- $u_{yy}(x_3, y_2)$
- $u_{xy}(x_2, y_3)$

Lecture 28

The Main Sources of Error

Truncation Error

Truncation error is defined as the error caused directly by an approximation method. For instance, all numerical integration methods are approximations and so there is error, even if the calculations are performed exactly. Numerical differentiation also has a truncation error, as will the differential equations methods we will study in Part IV, which are based on numerical differentiation formulas. There are two ways to minimize truncation error: (1) use a higher order method, and (2) use a finer grid so that points are closer together. Unless the grid is very small, truncation errors are usually much larger than roundoff errors. The obvious tradeoff is that a smaller grid requires more calculations, which in turn produces more roundoff errors and requires more running time.

Roundoff Error

Roundoff error always occurs when a finite number of digits are recorded after an operation. Fortunately, this error is extremely small. The standard measure of how small is called **machine epsilon**. It is defined as the smallest number that can be added to 1 to produce another number on the machine, i.e. if a smaller number is added the result will be rounded down to 1. In IEEE standard double precision (used by MATLAB and most serious software), machine epsilon is 2^{-52} or about 2.2×10^{-16} . A different, but equivalent, way of thinking about this is that the machine records 52 floating binary digits or about 15 floating decimal digits. Thus there are never more than 15 significant digits in any calculation. This of course is more than adequate for any application. However, there are ways in which this very small error can cause problems.

To see an unexpected occurrence of round-off try the following commands:

```
> (2^52+1) - 2^52  
> (2^53+1) - 2^53
```

Loss of Precision

One way in which small roundoff errors can become more significant is when significant digits cancel. For instance, if you were to calculate:

$$1234567(1 - .9999987654321)$$

then the result should be

$$1.52415678859930.$$

But, if you input:

```
> 1234567 * ( 1 - .9999987654321)
```

you will get:

```
> 1.52415678862573
```

In the correct calculation there are 15 significant digits, but the MATLAB calculation has only 11 significant digits.

Although this seems like a silly example, this type of *loss of precision* can happen by accident if you are not careful. For example in $f'(x) \approx (f(x+h) - f(x))/h$ you will lose precision when h gets too small. Try:

```
> format long
> f=inline('x^2','x')
> for i = 1:30
>     h=10^(-i)
>     df=(f(1+h)-f(1))/h
>     relerr=(2-df)/2
> end
```

At first the relative error decreases since truncation error is reduced. Then loss of precision takes over and the relative error increases to 1.

Bad Conditioning

We encountered bad conditioning in Part II, when we talked about solving linear systems. Bad conditioning means that the problem is unstable in the sense that small input errors can produce large output errors. This can be a problem in a couple of ways. First, the measurements used to get the inputs cannot be completely accurate. Second, the computations along the way have roundoff errors. Errors in the computations near the beginning especially can be magnified by a factor close to the condition number of the matrix. Thus what was a very small problem with roundoff can become a very big problem.

It turns out that matrix equations are not the only place where condition numbers occur. In any problem one can define the condition number as the maximum ratio of the relative errors in the output versus input, i.e.

$$\text{condition \# of a problem} = \max \left(\frac{\text{Relative error of output}}{\text{Relative error of inputs}} \right).$$

An easy example is solving a simple equation:

$$f(x) = 0.$$

Suppose that f' is close to zero at the solution x^* . Then a very small change in f (caused perhaps by an inaccurate measurement of some of the coefficients in f) can cause a large change in x^* . It can be shown that the condition number of this problem is $1/f'(x^*)$.

Exercises

28.1 ¹ The function $f(x) = (x - 2)^9$ could be evaluated as written, or first expanded as $f(x) = x^9 + 18x^8 + \dots$ and then evaluated. To find the expanded version, type

```
> syms x
> expand((x-2)^9)
> clear
```

To evaluate it the first way, type

```
> f1=inline('(x-2).^9','x')
> x=1.92:.001:2.08;
> y1=f1(x);
> plot(x,y1,'blue')
```

To do it the second way, convert the expansion above to an inline function `f2` and then type

```
> y2=f2(x);
> hold on
> plot(x,y2,'red')
```

Carefully study the resulting graphs. Should the graphs be the same? Which is more correct? MATLAB does calculations using approximately 16 decimal places. What is the largest error in the graph, and how big is it relative to 10^{-16} ? Which source of error is causing this problem?

¹From *Numerical Linear Algebra* by L. Trefethen and D. Baum, SIAM, 1997.

Review of Part III

Methods and Formulas

Polynomial Interpolation:

An exact fit to the data.

For n data points it is a $n - 1$ degree polynomial.

Only good for very few, accurate data points.

The coefficients are found by solving a linear system of equations.

Spline Interpolation:

Fit a simple function between each pair of points.

Joining points by line segments is the most simple spline.

Cubic is by far the most common and important.

Cubic matches derivatives and second derivatives at data points.

Simply supported and clamped ends are available.

Good for more, but accurate points.

The coefficients are found by solving a linear system of equations.

Least Squares:

Makes a “close fit” of a simple function to all the data.

Minimizes the sum of the squares of the errors.

Good for noisy data.

The coefficients are found by solving a linear system of equations.

Interpolation vs. Extrapolation: Polynomials, Splines and Least Squares are generally used for *Interpolation*, fitting between the data. *Extrapolation*, i.e. making fits beyond the data, is much more tricky. To make predictions beyond the data, you must have knowledge of the underlying process, i.e. what the function should be.

Numerical Integration:

Left Endpoint:

$$L_n = \sum_{i=1}^n f(x_{i-1})\Delta x_i$$

Right Endpoint:

$$R_n = \sum_{i=1}^n f(x_i)\Delta x_i.$$

Trapezoid Rule:

$$T_n = \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} \Delta x_i.$$

Midpoint Rule:

$$M_n = \sum_{i=1}^n f(\bar{x}_i)\Delta x_i \quad \text{where} \quad \bar{x}_i = \frac{x_{i-1} + x_i}{2}.$$

Numerical Integration Rules with Even Spacing:

For even spacing: $\Delta x = \frac{b-a}{n}$ where n is the number of subintervals, then:

$$L_n = \Delta x \sum_{i=0}^{n-1} y_i = \frac{b-a}{n} \sum_{i=0}^{n-1} f(x_i),$$

$$R_n = \Delta x \sum_{i=1}^n y_i = \frac{b-a}{n} \sum_{i=1}^n f(x_i),$$

$$T_n = \Delta x (y_0 + 2y_1 + \dots + 2y_{n-1} + y_n) = \frac{b-a}{2n} (f(x_0) + 2f(x_1) + \dots + 2f(x_{n-1}) + f(x_n)),$$

$$M_n = \Delta x \sum_{i=1}^n \bar{y}_i = \frac{b-a}{n} \sum_{i=1}^n f(\bar{x}_i),$$

Simpson's rule:

$$\begin{aligned} S_n &= \Delta x (y_0 + 4y_1 + 2y_2 + 4y_3 + \dots + 2y_{n-2} + 4y_{n-1} + y_n) \\ &= \frac{b-a}{3n} (f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)). \end{aligned}$$

Area of a region:

$$A = - \oint_C y \, dx,$$

where C is the counter-clockwise curve around the boundary of the region. We can represent such a curve, by consecutive points on it, i.e. $\bar{x} = (x_0, x_1, x_2, \dots, x_{n-1}, x_n)$, and $\bar{y} = (y_0, y_1, y_2, \dots, y_{n-1}, y_n)$ with $(x_n, y_n) = (x_0, y_0)$. Applying the trapezoid method to the integral (21.4):

$$A = - \sum_{i=1}^n \frac{y_{i-1} + y_i}{2} (x_i - x_{i-1})$$

Accuracy of integration rules:

Right and Left endpoint are $O(\Delta)$

Trapezoid and Midpoint are $O(\Delta^2)$

Simpson is $O(\Delta^4)$

Double Integrals on Rectangles:**Centerpoint:**

$$C_{mn} = \sum_{i,j=1,1}^{m,n} f(c_{ij}) A_{ij}.$$

where

$$c_{ij} = \left(\frac{x_{i-1} + x_i}{2}, \frac{y_{j-1} + y_j}{2} \right).$$

Centerpoint – Evenly spaced:

$$C_{mn} = \Delta x \Delta y \sum_{i,j=1,1}^{m,n} z_{ij} = \frac{(b-a)(d-c)}{mn} \sum_{i,j=1,1}^{m,n} f(c_{ij}).$$

Four corners:

$$F_{mn} = \sum_{i,j=1,1}^{m,n} \frac{1}{4} (f(x_i, y_j) + f(x_i, y_{j+1}) + f(x_{i+1}, y_j) + f(x_{i+1}, y_{j+1})) A_{ij},$$

Four Corners – Evenly spaced:

$$\begin{aligned}
 F_{mn} &= \frac{A}{4} \left(\sum_{\text{corners}} f(x_i, y_j) + 2 \sum_{\text{edges}} f(x_i, y_j) + 4 \sum_{\text{interior}} f(x_i, y_j) \right) \\
 &= \frac{(b-a)(d-c)}{4mn} \sum_{i,j=1,1}^{m,n} W_{ij} f(x_i, y_j).
 \end{aligned}$$

where

$$W = \begin{pmatrix} 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 4 & 4 & 4 & \cdots & 4 & 4 & 2 \\ 1 & 2 & 2 & 2 & \cdots & 2 & 2 & 1 \end{pmatrix}.$$

Double Simpson:

$$S_{mn} = \frac{(b-a)(d-c)}{9mn} \sum_{i,j=1,1}^{m,n} W_{ij} f(x_i, y_j).$$

where

$$W = \begin{pmatrix} 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 2 & 8 & 4 & 8 & \cdots & 4 & 8 & 2 \\ 4 & 16 & 8 & 16 & \cdots & 8 & 16 & 4 \\ 1 & 4 & 2 & 4 & \cdots & 2 & 4 & 1 \end{pmatrix}.$$

Integration based on triangles:

- Triangulation: Dividing a region up into triangles.
- Triangles are suitable for odd-shaped regions.
- A triangulation is better if the triangles are nearly equilateral.

Three corners:

$$T_n = \sum_{i=1}^n \bar{f}_i A_i$$

where \bar{f} is the average of f at the corners of the i -th triangle.

Area of a triangle with corners (x_1, y_1) , (x_2, y_2) , (x_3, y_3) :

$$A = \frac{1}{2} \left| \det \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \right|.$$

Centerpoint:

$$C = \sum_{i=1}^n f(\bar{x}_i, \bar{y}_i) A_i, \quad \text{with}$$

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3} \quad \text{and} \quad \bar{y} = \frac{y_1 + y_2 + y_3}{3}.$$

Forward Difference:

$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

Backward Difference:

$$f'(x_i) \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}.$$

Central Difference:

$$f'(x_i) = y'_i \approx \frac{y_{i+1} - y_{i-1}}{2h}.$$

Higher order central differences:

$$\begin{aligned} f''(x_i) &= y''_i \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \\ f'''(x_i) &= y'''_i \approx \frac{1}{2h^3} [y_{i+2} - 2y_{i+1} + 2y_{i-1} - y_{i-2}], \\ f^{(4)}(x_i) &= y_i^{(4)} \approx \frac{1}{h^4} [y_{i+2} - 4y_{i+1} + 6y_i - 4y_{i-1} + y_{i-2}]. \end{aligned}$$

Partial Derivatives: Denote $u_{i,j} = u(x_i, y_j)$.

$$\begin{aligned} u_x(x_i, y_j) &\approx \frac{1}{2h} (u_{i+1,j} - u_{i-1,j}), \\ u_y(x_i, y_j) &\approx \frac{1}{2k} (u_{i,j+1} - u_{i,j-1}), \\ u_{xx}(x_i, y_j) &\approx \frac{1}{h^2} (u_{i+1,j} - 2u_{i,j} + u_{i-1,j}), \\ u_{yy}(x_i, y_j) &\approx \frac{1}{k^2} (u_{i,j+1} - 2u_{i,j} + u_{i,j-1}), \\ u_{xy}(x_i, y_j) &\approx \frac{1}{4hk} (u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}). \end{aligned}$$

Sources of error:

Truncation – the method is an approximation.

Roundoff – double precision arithmetic uses ≈ 15 significant digits.

Loss of precision – an amplification of roundoff error due to cancellations.

Bad conditioning – the problem is sensitive to input errors.

Error can build up after multiple calculations.

Matlab

Data Interpolation:

Use the plot command `plot(x,y,'*')` to plot the data.

Use the Basic Fitting tool to make an interpolation or spline.

If you choose an $n - 1$ degree polynomial with n data points the result will be the exact polynomial interpolation.

If you select a polynomial degree less than $n - 1$, then MATLAB will produce a least squares approximation.

Integration

> `quadl(f,a,b)` Numerical integral of $f(x)$ on $[a, b]$.

> `dblquad(f,a,b,c,d)` Integral of $f(x,y)$ on $[a,b] \times [c,d]$.

Example:

> `f = inline('sin(x.*y)/sqrt(x+y)', 'x', 'y')`

> `I = dblquad(f,0,1,0,2)`

MATLAB uses an advanced form of Simpson's method.

Integration over non-rectangles:

Redefine the function to be zero outside the region. For example:

> `f = inline('sin(x.*y).^3.*(2*x + y <= 2)')`

> `I = dblquad(f,0,1,0,2)`

Integrates $f(x,y) = \sin^3(xy)$ on the triangle with corners $(0,0)$, $(0,2)$, and $(1,0)$.

Triangles:

MATLAB stores triangulations as a matrix of vertices V and triangles T .

> `T = delaunay(V)` Produces triangles from vertices.

> `trimesh(T,x,y)`

> `trimesh(T,x,y,z)`

> `trisurf(T,x,y)`

> `trisurf(T,x,y,z)`