MATLAB for M152B

© 2009 Peter Howard

Spring 2009

MATLAB for M152B

P. Howard

Spring 2009

Contents

1 Numerical Methods for Integration, Part 1

In the previous section we used MATLAB's built-in function *quad* to approximate definite integrals that could not be evaluated by the Fundamental Theorem of Calculus. In order to understand the type of calculation carried out by *quad*, we will develop a variety of algorithms for approximating values of definite integrals. In this section we will introduce the Right Endpoint Method and the Midpoint Method, and in the next section we will develop the Trapezoidal Rule and Simpson's Rule. Whenever numerical methods are used to make an approximation it's important to have some measure of the size of the error that can occur, and for each of these methods we give upper bounds on the error.

1.1 Riemann Sums with Right Endpoints

In our section on the numerical evaluation of Riemann sums, we saw that a Riemann sum with n points is an approximation of the definite integral of a function. For example, if we take the partition $P = [x_0, x_1, \dots, x_n]$, and we evaluate our function at right endpoints, then

$$\int_a^b f(x)dx \simeq \sum_{k=1}^n f(x_k)\Delta x_k,$$

where the larger n is the better this approximation is. If we are going to employ this approximation in applications, we need to have a more quantitative idea of how close it is to the exact value of the integral. That is, we need an estimate on the error. We define the error for this approximation by

$$E_R := |\int_a^b f(x)dx - \sum_{k=1}^n f(x_k)\Delta x_k|.$$

In order to derive a bound on this error, we will assume (for reasons that will become apparent during the calculation) that the derivative of f is bounded. That is, we assume there is some constant M so that $|f'(x)| \le M$ for all $x \in [a, b]$. We now begin computing

$$\begin{aligned} E_R :=& |\int_a^b f(x)dx - \sum_{k=1}^n f(x_k)\Delta x_k| \\ =& |\sum_{k=1}^n \int_{x_{k-1}}^{x_k} f(x)dx - \sum_{k=1}^n f(x_k)\Delta x_k|, \end{aligned}$$

where we have used the relation

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x) + \cdots + \int_{x_{n-1}}^{x_n} f(x)dx = \sum_{k=1}^n \int_{x_{k-1}}^{x_k} f(x)dx,$$

which follows from a repeated application of the integration property

$$\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^d f(x)dx.$$

Observing also that

$$\int_{x_{k-1}}^{x_k} dx = (x_k - x_{k-1}) = \Delta x_k,$$

we see that

$$\begin{aligned} E_R &= \Big|\sum_{k=1}^{n} \int_{x_{k-1}}^{x_k} (f(x) - f(x_k))dx\Big| \\ &= \Big|\sum_{k=1}^{n} \int_{x_{k-1}}^{x_k} f'(c_k(x))(x - x_k)dx\Big|, \end{aligned}$$

where in this step we have used the Mean Value Theorem to write

$$f(x) - f(x_k) = f'(c_k(x))(x - x_k),$$

where $c_k \in [x, x_k]$ and so clearly depends on the changing value of x. One of our properties of sums and integrations is that if we bring the absolute value inside we get a larger value. (This type of inequality is typically called a *triangle* inequality.) In this case, we have

$$\begin{aligned} E_R &\le \sum_{k=1}^{n} \int_{x_{k-1}}^{x_k} |f'(c_k(x))(x - x_k)|dx \\ &\le \sum_{k=1}^{n} \int_{x_{k-1}}^{x_k} M(x_k - x)dx \\ &= M\sum_{k=1}^{n} \Big[-\frac{(x_k - x)^2}{2}\Big|_{x_{k-1}}^{x_k} \Big] \\ &= M\sum_{k=1}^{n} \frac{(x_k - x_{k-1})^2}{2} = \frac{M}{2}\sum_{k=1}^{n} \triangle x_k^2 = \frac{M}{2}\sum_{k=1}^{n} \frac{(b-a)^2}{n^2} \\ &= \frac{M}{2}\frac{(b-a)^2}{n^2}\sum_{k=1}^{n} 1 = \frac{M(b-a)^2}{2n}. \end{aligned}$$

That is, the error for this approximation is

$$E_R = \frac{M(b-a)^2}{2n}.$$

Error estimate for Riemann sums with right endpoints. Suppose $f(x)$ is continuously differentiable on the interval $[a, b]$ and that $|f'(x)| \le M$ on $[a, b]$ for some finite value M. Then the maximum error allowed by Riemann sums with evaluation at right endpoints is

$$E_R = \frac{M(b-a)^2}{2n}.$$

In order to understand what this tells us, we recall that we have previously used the M-file *rsum1.m* to approximate the integral

$$\int_0^2 e^{-x^2}dx,$$

which to four decimal places is .8821. When we took $n = 4$, our approximation was .6352, with an error of

$$.8821 - .6352 = .2458.$$

In evaluating E_R, we must find a bound on $|f'(x)|$ on the interval $[0, 2]$, where

$$f'(x) = -2xe^{-x^2}.$$

We can accomplish this with our methods of maximization and minimization from first semester; that is, by computing the derivative of $f'(x)$ and setting this to 0 etc. Keeping in mind that we are trying to maximize/minimize $f'(x)$ (rather than $f(x)$), we compute

$$f''(x) = -2e^{-x^2} + 4x^2e^{-x^2} = e^{-x^2}(-2 + 4x^2) = 0,$$

which has solutions

$$x = \pm\frac{1}{\sqrt{2}}.$$

The possible extremal points for $x \in [0, 2]$ are 0, $\frac{1}{\sqrt{2}}$, and 2, and we have

$$\begin{aligned} f'(0) =& 0 \\ f'(\frac{1}{\sqrt{2}}) =& -\sqrt{2}e^{-\frac{1}{2}} = -.8578 \\ f'(2) =& -4e^{-4} = -.0733. \end{aligned}$$

The maximum value of $|f'(x)|$ over this interval is $M = .8578$. Our error estimate guarantees that our error for $n = 4$ will be less than or equal to

$$\frac{.8578(2)^2}{8} = .4289.$$

We observe that the actual error (.2458) is much less than the maximum error (.4289), which isn't particularly surprising since the maximum error essentially assumes that $f'(x)$ will always be the largest it can be. Finally, suppose we would like to ensure that our approximation to this integral has an error smaller than .0001. We need only find n so that

$$\frac{M(b-a)^2}{2n} < .0001.$$

That is, we require

$$n > \frac{M(b-a)^2}{2(.0001)}.$$

For this example,

$$n > \frac{.8578(4)}{.0002} = 17,156.$$

We can easily verify that this is sufficient with the following MATLAB calculation.

```
>>f=inline('exp(-x^2)');
>>rsum1(f,0,2,17157)
ans =
0.8821
```

1.2 Riemann Sums with Midpoints (The Midpoint Rule)

In our section on the numerical evaluation of Riemann sums, we saw in the homework that one fairly accurate way in which to approximate the value of a definite integral was to use a Riemann sum with equally spaced subintervals and to evaluate the function at the midpoint of each interval. This method is called the midpoint rule. Since writing an algorithm for the midpoint rule is a homework assignment, we won't give it here, but we do provide an error estimate.

Error estimate for the Midpoint Rule. Suppose $f(x)$ is twice continuously differentiable on the interval $[a, b]$ and that $|f''(x)| \leq M$ for some finite value M. Then the maximum error allowed by the midpoint rule is

$$E_M = \frac{M(b-a)^3}{24n^2}.$$

The proof of this result is similar to the one given above for evaluation at right endpoints, except that it requires a generalization of the Mean Value Theorem; namely Taylor approximation with remainder. Since Taylor approximation is covered later in the course, we won't give a proof of this estimate here.

1.3 Assignments

1. [3 pts] For the integral

$$\int_1^5 \sqrt{1+x^3}dx$$

find the maximum possible error that will be obtained with a Riemann sum approximation that uses 100 equally spaced subintervals and right endpoints. Use *rsum1.m* (from Section 13, Fall semester) to show that the error is less than this value. (You can use *quad* to get a value for this integral that is exact to four decimal places.)

2. [3 pts] For the integral from Problem 1, use E_R to find a number of subintervals n that will ensure that approximation by Riemann sums with right endpoints will have an error less than .01. Use *rsum1.m* to show that this number n works.

3. [4 pts] For the integral

$$\int_0^2 e^{-x^2}dx,$$

use E_M to find a number of subintervals n that guarantees an error less than .0001 when the approximation is carried out by the midpoint rule. Compare this value of n with the one obtained above for this integration with right endpoints. Use your M-file from Problem 2 of Section 13 from Fall semester to show that this value of n works.

2 Numerical Methods for Integration, Part 2

In addition to Riemann sums, a number of algorithms have been developed for numerically evaluating definite integrals. In these notes we will consider the two most commonly discussed in introductory calculus classes: the Trapezoidal Rule and Simpson's Rule.

2.1 The Trapezoidal Rule

If we approximate the integral $\int_a^b f(x)dx$ with a Riemann sum that has subintervals of equal width and evaluation at right endpoints, we have

$$\int_a^b f(x)dx \approx \sum_{k=1}^n f(x_k)\Delta x_k.$$

On the other hand, if we evaluate at left endpoints, we obtain

$$\int_a^b f(x)dx \approx \sum_{k=1}^n f(x_{k-1})\Delta x_k.$$

One thing we have observed in several examples is that if a function is increasing then approximation by evaluation at right endpoints always overestimates the area, while approximation by evaluation at left endpoints underestimates the area. (Precisely the opposite is true for decreasing functions.) In order to address this, the Trapezoidal Rule takes an average of these two approximations. That is, the Trapezoidal Rule approximates $\int_a^b f(x)dx$ with (recall: $\Delta x_k = \frac{b-a}{n}$)

$$\begin{aligned}
T_n &= \frac{\sum_{k=1}^n f(x_k)\Delta x_k + \sum_{k=1}^n f(x_{k-1})\Delta x_k}{2} \\
&= \frac{b-a}{2n}\sum_{k=1}^n (f(x_k) + f(x_{k-1})) \\
&= \frac{b-a}{2n}\Big(f(x_0) + 2\sum_{k=1}^{n-1} f(x_k) + f(x_n)\Big) \\
&= \frac{b-a}{n}\Big(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(x_k)\Big).
\end{aligned}$$

(The name arises from the observation that this is precisely the approximation we would arrive at if we partitioned the interval $[a,b]$ in the usual way and then approximated the area under $f(x)$ on each subinterval $[x_{k-1}, x_k]$ with trapezoids with sidelengths $f(x_{k-1})$ and $f(x_k)$ rather than with rectangles.) We can carry such a calculation out in MATLAB with the M-file *trap.m*.

```
function value=trap(f,a,b,n)
%TRAP: Approximates the integral of f from a to b
%using the Trapezoidal Rule with n subintervals
```

```
%of equal width.
value = (f(a)+f(b))/2;
dx = (b-a)/n;
for k=1:(n-1)
c = a+k*dx;
value = value + f(c);
end
value = dx*value;
```

In order to compare this with previous methods, we use *trap.m* to approximate the integral

$$\int_0^2 e^{-x^2} dx.$$

```
>>f=inline('exp(-x^2)')
f =
Inline function:
f(x) = exp(-x^2)
>>trap(f,0,2,10)
ans =
0.8818
>>trap(f,0,2,100)
ans =
0.8821
>>trap(f,0,2,1000)
ans =
0.8821
```

By comparing these numbers with those obtained for this same approximation with evaluation at right endpoints and evaluation at midpoints (the Midpoint Rule), we see that the Trapezoidal Rule is certainly more accurate than evaluation at right endpoints and is roughly comparable to the Midpoint Rule. In order to be more specific about the accuracy of the Trapezoidal Rule, we give, without proof, an error estimate.

Error estimate for the Trapezoidal Rule. Suppose $f(x)$ is twice continuously differentiable on the interval $[a, b]$ and that $|f''(x)| \leq M$ for some finite value M. Then the maximum error allowed by the Trapezoidal Rule is

$$E_T = \frac{M(b-a)^3}{12n^2}.$$

2.2 Simpson's Rule

The final method we will consider for the numerical approximation of definite integrals is known as Simpson's Rule after a self-taught 18th century English mathematician named Thomas Simpson (1710–1761), who did not invent the method, but did popularize it in his book *A New Treatise of Fluxions.*[1] In the previous section we derived the Trapezoidal Rule as an average of right and left Riemann sums, but we could have proceeded instead by suggesting at the outset that we approximate $f(x)$ on each interval $[x_{k-1}, x_k]$ by the line segment connecting the points $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$, thereby giving the trapezoids that give the method its name. The idea behind Simpson's Rule is to proceed by approximating $f(x)$ by quadratic polynomials rather than by linear ones (lines). More generally, any order of polynomial can be used to approximate f, and higher orders give more accurate methods.

Before considering Simpson's Rule, recall that for any three points (x_1, y_1), (x_2, y_2), and (x_3, y_3) there is exactly one polynomial

$$p(x) = ax^2 + bx + c$$

that passes through all three points. (If the points are all on the same line, we will have $a = 0$.) In particular, the values of a, b, and c satisfy the system of equations

$$\begin{aligned} y_1 &= ax_1^2 + bx_1 + c \\ y_2 &= ax_2^2 + bx_2 + c \\ y_3 &= ax_3^2 + bx_3 + c, \end{aligned}$$

which can be solved so long as

$$(x_1 - x_3)(x_1 - x_2)(x_2 - x_3) \neq 0.$$

That is, so long as we have three different x-values. (This follows from a straightforward linear algebra calculation, which we omit here but will cover later in the course.)

Suppose now that we would like to approximate the integral

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

Let P be a partition of $[a, b]$ with the added restriction that P consists of an even number of points (i.e., n is an even number). From the discussion in the previous paragraph, we see that we can find a quadratic function $p(x) = a_1x^2 + b_1x + c_1$ that passes through the first three points $(x_0, f(x_0))$, $(x_1, f(x_1))$, and $(x_2, f(x_2))$. Consequently, we can approximate the area below the graph of $f(x)$ from x_0 to x_2 by

$$\int_{x_0}^{x_2} (a_1x^2 + b_1x + c_1)dx = \frac{a_1}{3}x^3 + \frac{b_1}{2}x^2 + c_1x\Big|_{x_0}^{x_2} = (\frac{a_1}{3}x_2^3 + \frac{b_1}{2}x_2^2 + c_1x_2) - (\frac{a_1}{3}x_0^3 + \frac{b_1}{2}x_0^2 + c_1x_0).$$

We can then approximate the area between x_2 and x_4 by similarly fitting a polynomial through the three points $(x_2, f(x_2))$, $(x_3, f(x_3))$, and $(x_4, f(x_4))$. More generally, for every odd value of k, we will fit a polynomial through the three points $(x_{k-1}, f(x_{k-1}))$, $(x_k, f(x_k))$, and $(x_{k+1}, f(x_{k+1}))$. Recalling that $x_{k-1} = x_k - \Delta x$ and $x_{k+1} = x_k + \Delta x$, we see that the area in this general case is

$$A_k = \int_{x_k-\Delta x}^{x_k+\Delta x} a_kx^2 + b_kx + c_kdx = 2\Delta x(\frac{1}{3}a_k(\Delta x)^2 + a_kx_k^2 + b_kx_k + c_k) \tag{1}$$

[1]Isaac Newton referred to derivatives as *fluxions,* and most British authors in the 18th century followed his example.

(see homework assignments). As specified in the previous paragraph, we can choose a_k, b_k and c_k to solve the system of three equations

$$\begin{aligned} f(x_{k-1}) &= a_k(x_k - \Delta x)^2 + b_k(x_k - \Delta x) + c_k \\ f(x_k) &= a_k x_k^2 + b_k x_k + c_k \\ f(x_{k+1}) &= a_k(x_k + \Delta x)^2 + b_k(x_k + \Delta x) + c_k. \end{aligned}$$

From these equations, we can see the relationship

$$f(x_{k-1}) + 4f(x_k) + f(x_{k+1}) = 6(\frac{1}{3}a_k(\Delta x)^2 + a_k x_k^2 + b_k x_k + c_k). \tag{2}$$

Comparing (2) with (1), we see that

$$A_k = \frac{\Delta x}{3}(f(x_{k-1}) + 4f(x_k) + f(x_{k+1})).$$

Combining these observations, we have the approximation relation

$$\int_a^b f(x)dx \approx A_1 + A_3 + \cdots + A_{n-1},$$

or equivalently

$$\int_a^b f(x)dx \approx \frac{\triangle x}{3}\Big(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\Big).$$

Error estimate for Simpson's Rule. Suppose $f(x)$ is four times continuously differentiable on the interval $[a, b]$ and that $|f^{(4)}(x)| \le M$ for some finite value M. Then the maximum error allowed by Simpson's Rule is

$$E_S = \frac{M(b-a)^5}{180n^4}.$$

As a final remark, I'll mention that Simpson's Rule can also be obtained in the following way: Observing that for functions that are either concave up or concave down the midpoint rule underestimates error while the Trapezoidal Rule overestimates it, we might try averaging the two methods. If we let M_n denote the midpoint rule with n points and we let T_n denote the Trapezoidal Rule with n points, then the weighted average $\frac{1}{3}T_n + \frac{2}{3}M_n$ is Simpson's Rule with $2n$ points. (Keep in mind that for a given partition the Trapezoidal Rule and the Midpoint Rule use different points, so if the partition under consideration has n points then the expression $\frac{1}{3}T_n + \frac{2}{3}M_n$ involves evaluation at twice this many points.)

2.3 Assignments

1. [3 pts] Show that (1) is correct.

2. [3 pts] Use E_S to find a number of subintervals n that will ensure that approximation of $\int_0^2 e^{-x^2}dx$ by Simpson's Rule will have an error less than .0001.

3. [4 pts] Write a MATLAB M-file similar to *trap.m* that approximates definite integrals with Simpson's Rule. Use your M-file to show that your value of n from Problem 2 works.

3 Taylor Series in MATLAB

3.1 Taylor Series

First, let's review our two main statements on Taylor polynomials with remainder.

Theorem 3.1. (Taylor polynomial with integral remainder) Suppose a function $f(x)$ and its first $n+1$ derivatives are continuous in a closed interval $[c,d]$ containing the point $x=a$. Then for any value x on this interval

$$f(x) = f(a) + f'(a)(x-a) + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \frac{1}{n!}\int_a^x f^{(n+1)}(y)(x-y)^n dy.$$

Theorem 3.2. (Generalized Mean Value Theorem) Suppose a function $f(x)$ and its first n derivatives are continuous in a closed interval $[a,b]$, and that $f^{(n+1)}(x)$ exists on (a,b). Then there exists some $c \in (a,b)$ so

$$f(b) = f(a) + f'(a)(b-a) + \cdots + \frac{f^{(n)}(a)}{n!}(b-a)^n + \frac{f^{(n+1)}(c)}{(n+1)!}(b-a)^{n+1}.$$

It's clear from the fact that $n!$ grows rapidly as n increases that for sufficiently differentiable functions $f(x)$ Taylor polynomials become more accurate as n increases.

Example 3.1. Find the Taylor polynomials of orders 1, 3, 5, and 7 near $x=0$ for $f(x)=\sin x$. (Even orders are omitted because Taylor polynomials for $\sin x$ have no even order terms.

The MATLAB command for a Taylor polynomial is *taylor(f,n+1,a)*, where f is the function, a is the point around which the expansion is made, and n is the order of the polynomial. We can use the following code:

```
>>syms x
>>f=inline('sin(x)')
f =
Inline function:
f(x) = sin(x)
>>taylor(f(x),2,0)
ans =
x
>>taylor(f(x),4,0)
ans =
x-1/6*x^3
>>taylor(f(x),6,0)
ans =
x-1/6*x^3+1/120*x^5
>>taylor(f(x),8,0)
ans =
x-1/6*x^3+1/120*x^5-1/5040*x^7
```

△

Example 3.2. Find the Taylor polynomials of orders 1, 2, 3, and 4 near $x = 1$ for $f(x) = \ln x$.

In MATLAB:

```
>>syms x
>>f=inline('log(x)')
f =
Inline function:
f(x) = log(x)
>>taylor(f(x),2,1)
ans =
x-1
>>taylor(f(x),3,1)
ans =
x-1-1/2*(x-1)^2
>>taylor(f(x),4,1)
ans =
x-1-1/2*(x-1)^2+1/3*(x-1)^3
>>taylor(f(x),5,1)
ans =
x-1-1/2*(x-1)^2+1/3*(x-1)^3-1/4*(x-1)^4
```

△

Example 3.3. For $x \in [0, \pi]$, plot $f(x) = \sin x$ along with Taylor approximations around $x = 0$ with $n = 1, 3, 5, 7$.

We can now solve Example 1 with the following MATLAB function M-file, *taylorplot.m.*

```
function taylorplot(f,a,left,right,n)
%TAYLORPLOT: MATLAB function M-file that takes as input
%a function in inline form, a center point a, a left endpoint,
%a right endpoint, and an order, and plots
%the Taylor polynomial along with the function at the given order.
syms x
p = vectorize(taylor(f(x),n+1,a));
x=linspace(left,right,100);
f=f(x);
p=eval(p);
plot(x,f,x,p,'r')
```

The plots in Figure 1 can now be created with the sequence of commands

```
>>f=inline('sin(x)')
f =
Inline function:
```

f(x) = sin(x)
>>taylorplot(f,0,0,pi,1)
>>taylorplot(f,0,0,pi,3)
>>taylorplot(f,0,0,pi,5)
>>taylorplot(f,0,0,pi,7)

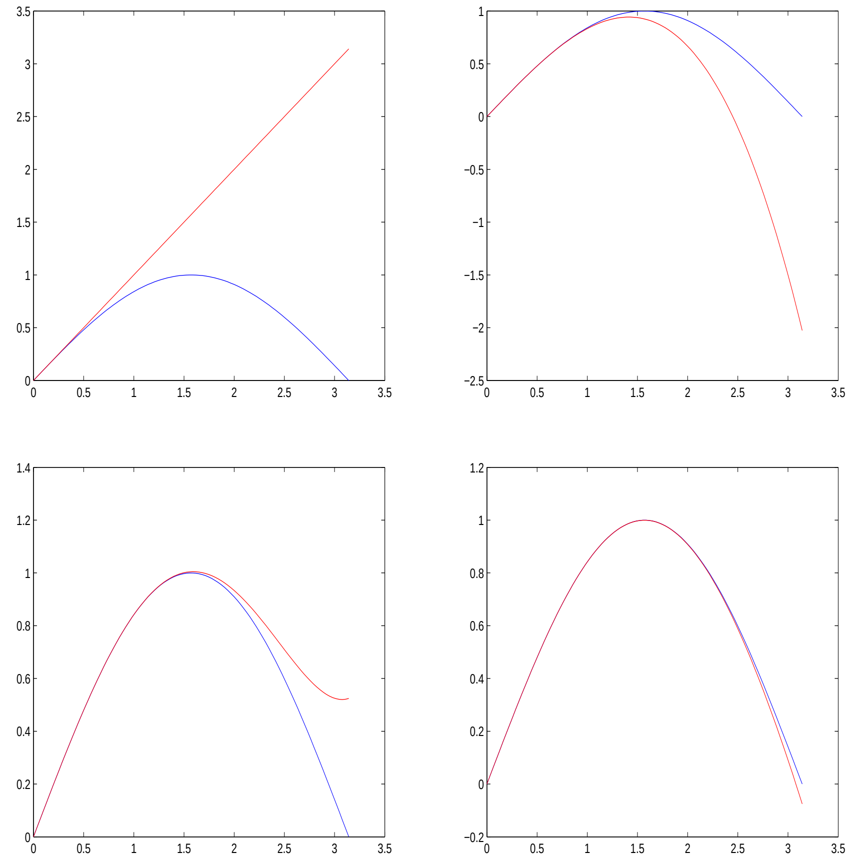


Figure 1: Taylor approximations of $f(x) = \sin x$ with $n = 1, 3, 5, 7$.

$\triangle$

Example 3.4. Use a Taylor polynomial around $x = 0$ to approximate the natural base e with an accuracy of .0001.

First, we observe that for $f(x) = e^x$, we have $f^{(k)}(x) = e^x$ for all $k = 0, 1, \ldots$. Consequently, the Taylor expansion for e^x is

$$e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \ldots.$$

If we take an n^{th} order approximation, the error is

$$\frac{f^{(n+1)}(c)}{(n+1)!}x^{n+1},$$

where $c \in (a, b)$. Taking $a = 0$ and $b = 1$ this is less than

$$\sup_{c\in(0,1)} \frac{e^c}{(n+1)!} = \frac{e}{(n+1)!} \leq \frac{3}{(n+1)!}.$$

Notice that we are using a crude bound on e, because if we are trying to estimate it, we should not assume we know its value with much accuracy. In order to ensure that our error is less than .0001, we need to find n so that

$$\frac{3}{(n+1)!} < .0001.$$

Trying different values for n in MATLAB, we eventually find

```
>>3/factorial(8)
ans =
7.4405e-05
```

which implies that the maximum error for a 7th order polynomial with $a = 0$ and $b = 1$ is .000074405. That is, we can approximate e with

$$e^1 = 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!} = 2.71825,$$

which we can compare with the correct value to five decimal places

$$e = 2.71828.$$

The error is $2.71828 - 2.71825 = .00003$. △

3.2 Partial Sums in MATLAB

For the infinite series $\sum_{k=1}^{\infty} a_k$, we define the n^{th} partial sum as

$$S_n := \sum_{k=1}^{n} a_k.$$

We say that the infinite series converges to S if

$$\lim_{n\to\infty} S_n = S.$$

Example 3.5. For the series

$$\sum_{k=1}^{\infty} \frac{1}{k^{3/2}} = 1 + \frac{1}{2^{3/2}} + \frac{1}{3^{3/2}} + \dots,$$

compute the partial sums S_{10}, S_{10000}, $S_{1000000}$, and $S_{10000000}$ and make a conjecture as to whether or not the series converges, and if it converges, what it converges to.

We can solve this with the following MATLAB code:

```
>>k=1:10;
>>s=sum(1./k.^(3/2))
s =
1.9953
>>k=1:10000;
>>s=sum(1./k.^(3/2))
s =
2.5924
>>k=1:1000000;
s=sum(1./k.^(3/2))
s =
2.6104
>>k=1:10000000;
s=sum(1./k.^(3/2))
s =2.6117
```

The series seems to be converging to a value near 2.61. △

Example 3.6. For the harmonic series

$$\sum_{k=1}^{\infty} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \dots,$$

compute the partial sums S_{10}, S_{10000}, $S_{1000000}$, and $S_{10000000}$, and make a conjecture as to whether or not the series converges, and if it converges, what it converges to. We can solve this with the following MATLAB code:

```
>>k=1:10;
>>s=sum(1./k)
s =
2.9290
>>k=1:10000;
>>s=sum(1./k)
s =
9.7876
>>k=1:1000000;
>>s=sum(1./k)
s =
14.3927
>>k=1:10000000;
>>s=sum(1./k)
s =
16.6953
```

As we will show in class, this series diverges. △

3.3 Assignments

1. [3 pts] For $x \in [0, \pi]$, plot $f(x) = \cos x$ along with Taylor approximations around $x = 0$ with $n = 0, 2, 4, 6$.

2. [4 pts] Use an appropriate Taylor series to write a MATLAB function M-file that approximates $f(x) = \sin(x)$ for all x with a maximum error of .01. That is, your M-file should take values of x as input, return values of $\sin x$ as output, but should *not* use MATLAB's built-in function *sin.m*. (Hint. You might find MATLAB's built-in file *mod* useful in dealing with the periodicity.)

3. [3 pts] Use numerical experiments to arrive at a conjecture about whether or not the infinite series

$$\sum_{k=2}^{\infty} \frac{1}{k(\ln k)^2}$$

converges.

4 Solving ODE Symbolically in MATLAB

4.1 First Order Equations

We can solve ordinary differential equations symbolically in MATLAB with the built-in M-file *dsolve.*

Example 4.1. Find a general solution for the first order differential equation

$$y'(x) = yx. \tag{3}$$

We can accomplish this in MATLAB with the following single command, given along with MATLAB's output.

```
>>y = dsolve('Dy = y*x','x')
y = C1*exp(1/2*x^2)
```

Notice in particular that MATLAB uses capital D to indicate the derivative and requires that the entire equation appear in single quotes. MATLAB takes t to be the independent variable by default, so here x must be explicitly specified as the independent variable. Alternatively, if you are going to use the same equation a number of times, you might choose to define it as a variable, say, *eqn1.*

```
>>eqn1 = 'Dy = y*x'
eqn1 =
Dy = y*x
>>y = dsolve(eqn1,'x')
y = C1*exp(1/2*x^2)
```

△

Example 4.2. Solve the initial value problem

$$y'(x) = yx; \quad y(1) = 1. \tag{4}$$

Again, we use the *dsolve* command.

```
>>y = dsolve(eqn1,'y(1)=1','x')
y =
1/exp(1/2)*exp(1/2*x^2)
```

or

```
>>inits = 'y(1)=1';
>>y = dsolve(eqn1,inits,'x')
y =
1/exp(1/2)*exp(1/2*x^2)
```

△

Now that we've solved the ODE, suppose we would like to evaluate the solution at specific values of x. In the following code, we use the *subs* command to evaluate $y(2)$.

```
>>x=2;
>>subs(y)
ans =
4.4817
```

Alternatively, we can define $y(x)$ as an inline function, and then evaluate this inline function at the appropriate value.

```
>>yf=inline(char(y))
yf =
Inline function:
yf(x) = 1/exp(1/2)*exp(1/2*x^2)
>>yf(2)
ans =
4.4817
```

On the other hand, we might want to find the value x that corresponds with a given value of y (i.e., invert $y(x)$). For example, suppose we would like to find the value of x at which $y(x) = 2$. In this case, we want to solve

$$2 = e^{-\frac{1}{2}+\frac{x^2}{2}}.$$

In MATLAB:

```
>>solve(y-2)
ans =
(2*log(2)+1)^(1/2)
-(2*log(2)+1)^(1/2)
```

As expected (since x appears squared), we see that two different values of x give $y = 2$. (Recall that the *solve* command sets its argument to 0.)

Finally, let's sketch a graph of $y(x)$. In doing this, we run immediately into two minor difficulties: (1) our expression for $y(x)$ isn't suited for array operations (.*, ./, .^), and (2) y, as MATLAB returns it, is actually a symbol (a *symbolic object*). The first of these obstacles is straightforward to fix, using *vectorize()*. For the second, we employ the useful command *eval()*, which evaluates or executes text strings that constitute valid MATLAB commands. Hence, we can use the following MATLAB code to create Figure 2.

```
>>x = linspace(0,1,20);
>>y = eval(vectorize(y));
>>plot(x,y)
```

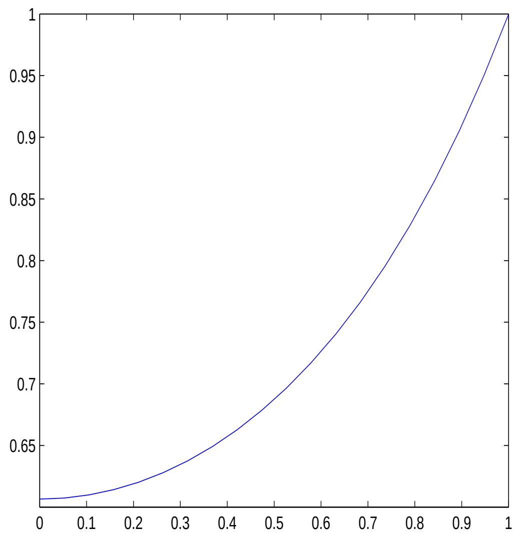


Figure 2: Plot of $y(x) = e^{-1/2}e^{\frac{1}{2}x^2}$ for $x \in [1, 20]$.

4.2 Second and Higher Order Equations

Example 4.3. Solve the second order differential equation

$$y''(x) + 8y'(x) + 2y(x) = \cos(x); \qquad y(0) = 0, \, y'(0) = 1, \tag{5}$$

and plot its solution for $x \in [0, 5]$.

The following MATLAB code suffices to create Figure 3.

```
>>eqn2 = 'D2y + 8*Dy + 2*y = cos(x)';
>>inits2 = 'y(0)=0, Dy(0)=1';
>>y=dsolve(eqn2,inits2,'x')
y =
exp((-4+14^(1/2))*x)*(53/1820*14^(1/2)-1/130)
+exp(-(4+14^(1/2))*x)*(-53/1820*14^(1/2)-1/130)+1/65*cos(x)+8/65*sin(x)
>>x=linspace(0,5,100);
>>y=eval(vectorize(y));
>>plot(x,y)
```

△

4.3 First Order Systems

Suppose we want to solve and plot solutions to the system of three ordinary differential equations

$$\begin{aligned} x'(t) &= x(t) + 2y(t) - z(t) \\ y'(t) &= x(t) + z(t) \\ z'(t) &= 4x(t) - 4y(t) + 5z(t). \end{aligned} \tag{6}$$

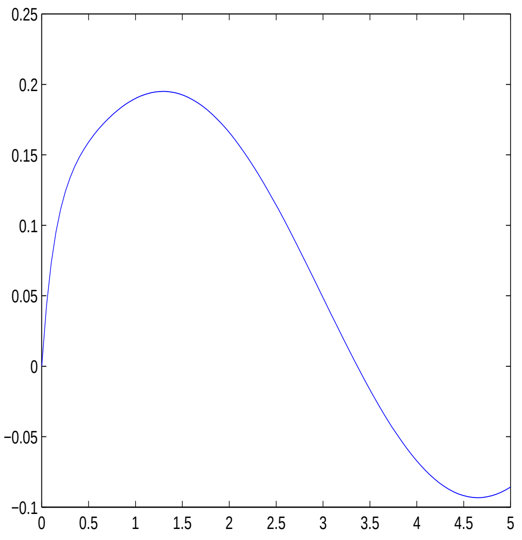


Figure 3: Plot of $y(x)$ for Example 4.3.

First, to find a general solution, we proceed similarly as in the case of single equations, except with each equation now braced in its own pair of (single) quotation marks:

```
>>[x,y,z]=dsolve('Dx=x+2*y-z','Dy=x+z','Dz=4*x-4*y+5*z')
x =
-C1*exp(3*t)-C2*exp(t)-2*C3*exp(2*t)
y =
C1*exp(3*t)+C2*exp(t)+C3*exp(2*t)
z =
4*C1*exp(3*t)+2*C2*exp(t)+4*C3*exp(2*t)
```

Notice that since no independent variable was specified, MATLAB used its default, t. To solve an initial value problem, we simply define a set of initial values and add them at the end of our *dsolve()* command. Suppose we have $x(0) = 1$, $y(0) = 2$, and $z(0) = 3$. We have, then,

```
>>inits='x(0)=1,y(0)=2,z(0)=3';
>>[x,y,z]=dsolve('Dx=x+2*y-z','Dy=x+z','Dz=4*x-4*y+5*z',inits)
x =
-5/2*exp(3*t)-5/2*exp(t)+6*exp(2*t)
y =
5/2*exp(3*t)+5/2*exp(t)-3*exp(2*t)
z =
10*exp(3*t)+5*exp(t)-12*exp(2*t)
```

Finally, we can create Figure 4 (a plot of the solution to this system for $t \in [0, .5]$) with the following MATLAB commands.

```
>>t=linspace(0,.5,25);
>>x=eval(vectorize(x));
>>y=eval(vectorize(y));
>>z=eval(vectorize(z));
>>plot(t, x, t, y, '–',t, z,':')
```

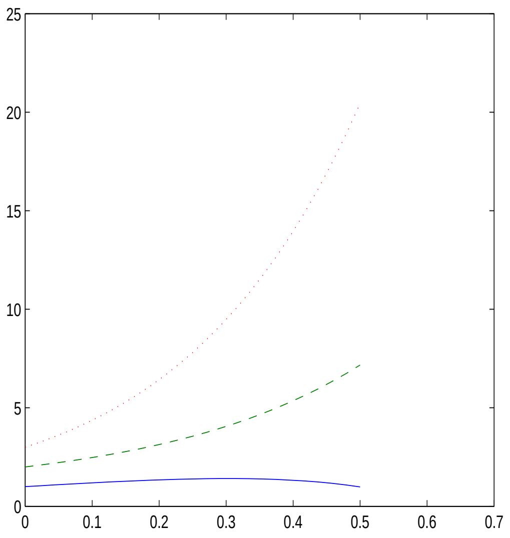


Figure 4: Solutions to equation (6).

4.4 Assignments

1. [2 pts] Find a general solution for the differential equation

$$\frac{dy}{dx} = e^y \sin x.$$

2. [2 pts] Solve the initial value problem

$$\frac{dy}{dx} = e^y \sin x; \quad y(2) = 3,$$

and evaluate $y(5)$. Find the value of x (in decimal form) associated with $y = 1$.
3. [2 pts] Solve the initial value problem

$$\frac{dp}{dt} = rp(1 - \frac{p}{K}); \quad p(0) = p_0.$$

4. [2 pts] Solve the initial value problem

$$7y'' + 2y' + y = x; \quad y(0) = 1, y'(0) = 2,$$

and plot your solution for $x \in [0,1]$.

5. [2 pts] Solve the system of differential equations

$$\begin{aligned} x'(t) &= 2x(t) + y(t) - z(t) \\ y'(t) &= x(t) + 5z(t) \\ z'(t) &= x(t) - y(t) + z(t), \end{aligned}$$

with $x(0) = 1$, $y(0) = 2$, and $z(0) = 3$ and plot your solution for $t \in [0,1]$.

5 Numerical Methods for Solving ODE

MATLAB has several built-in functions for approximating the solution to a differential equation, and we will consider one of these in Section 9. In order to gain some insight into what these functions do, we first derive two simple methods, Euler's method and the Taylor method of order 2.

5.1 Euler's Method

Consider the general first order differential equation

$$\frac{dy}{dx} = f(x,y); \quad y(x_0) = y_0, \tag{7}$$

and suppose we would like to solve this equation on the interval of x-values $[x_0, x_n]$. Our goal will be to approximate the value of the solution $y(x)$ at each of the x values in a partition $P = [x_0, x_1, x_2, ..., x_n]$. Since $y(x_0)$ is given, the first value we need to estimate is $y(x_1)$. By Taylor's Theorem, we can write

$$y(x_1) = y(x_0) + y'(x_0)(x_1 - x_0) + \frac{y'(c)}{2}(x_1 - x_0)^2,$$

where $c \in (x_0, x_1)$. Observing from our equation that $y'(x_0) = f(x_0, y(x_0))$, we have

$$y(x_1) = y(x_0) + f(x_0, y(x_0))(x_1 - x_0) + \frac{y'(c)}{2}(x_1 - x_0)^2.$$

If our partition P has small subintervals, then $x_1 - x_0$ will be small, and we can regard the smaller quantity $\frac{y'(c)}{2}(x_1 - x_0)^2$ as an error term. That is, we have

$$y(x_1) \approx y(x_0) + f(x_0, y(x_0))(x_1 - x_0). \tag{8}$$

We can now compute $y(x_2)$ in a similar manner by using Taylor's Theorem to write

$$y(x_2) = y(x_1) + y'(x_1)(x_2 - x_1) + \frac{y'(c)}{2}(x_2 - x_1)^2.$$

Again, we have from our equation that $y'(x_1) = f(x_1, y(x_1))$, and so

$$y(x_2) = y(x_1) + f(x_1, y(x_1))(x_2 - x_1) + \frac{y'(c)}{2}(x_2 - x_1)^2.$$

If we drop the term $\frac{y'(c)}{2}(x_2 - x_1)^2$ as an error, then we have

$$y(x_2) \approx y(x_1) + f(x_1, y(x_1))(x_2 - x_1),$$

where the value $y(x_1)$ required here can be approximated by the value from (8). More generally, for any $k = 1, 2, ..., n-1$ we can approximate $y(x_{k+1})$ from the relation

$$y(x_{k+1}) \approx y(x_k) + f(x_k, y(x_k))(x_{k+1} - x_k),$$

where $y(x_k)$ will be known from the previous calculation. As with methods of numerical integration, it is customary in practice to take our partition to consist of subintervals of equal width,

$$(x_{k+1} - x_k) = \Delta x = \frac{x_n - x_0}{n}.$$

(In the study of numerical methods for differential equations, this quantity is often denoted h.) In this case, we have the general relationship

$$y(x_{k+1}) \approx y(x_k) + f(x_k, y(x_k))\Delta x.$$

If we let the values $y_0, y_1, ..., y_n$ denote our approximations for y at the points $x_0, x_1, ..., x_n$ (that is, $y_0 = y(x_0)$, $y_1 \approx y(x_1)$, etc.), then we can approximate $y(x)$ on the partition P by iteratively computing

$$y_{k+1} = y_k + f(x_k, y_k)\Delta x. \tag{9}$$

Example 5.1. Use Euler's method (9) with $n = 10$ to approximate the solution to the differential equation

$$\frac{dy}{dx} = \sin(xy); \quad y(0) = \pi,$$

on the interval $[0, 1]$.

We will carry out the first few iterations in detail, and then we will write a MATLAB M-file to carry it out in its entirety. First, the initial value $y(0) = \pi$ gives us the values $x_0 = 0$ and $y_0 = \pi$. If our partition is composed of subintervals of equal width, then $x_1 = \triangle x = \frac{1}{10} = .1$, and according to (9)

$$y_1 = y_0 + \sin(x_0 y_0)\triangle x = \pi + \sin(0).1 = \pi.$$

We now have the point $(x_1, y_1) = (.1, \pi)$, and we can use this and (9) to compute

$$y_2 = y_1 + \sin(x_1 y_1)\triangle x = \pi + \sin(.1\pi)(.1) = 3.1725.$$

We now have $(x_2, y_2) = (.2, 3.1725)$, and we can use this to compute

$$y_3 = y_2 + \sin(x_2 y_2)\triangle x = 3.1725 + \sin(.2(3.1725))(.1) = 3.2318.$$

More generally, we can use the M-file *euler.m*.

```
function [xvalues, yvalues] = euler(f,x0,xn,y0,n)
%EULER: MATLAB function M-file that solves the
%ODE y'=f, y(x0)=y0 on [x0,y0] using a partition
%with n equally spaced subintervals
dx = (xn-x0)/n;
x(1) = x0;
y(1) = y0;
for k=1:n
x(k+1)=x(k) + dx;
y(k+1)= y(k) + f(x(k),y(k))*dx;
end
xvalues = x';
yvalues = y';
```

We can implement this file with the following code, which creates Figure 5.

```
>>f=inline('sin(x*y)')
f =
Inline function:
f(x,y) = sin(x*y)
>>[x,y]=euler(f,0,1,pi,10)
x =
0
0.1000
0.2000
0.3000
0.4000
0.5000
0.6000
0.7000
0.8000
0.9000
1.0000
y =
3.1416
3.1416
3.1725
3.2318
3.3142
3.4112
3.5103
3.5963
3.6548
3.6764
3.6598
>>plot(x,y)
>>[x,y]=euler(f,0,1,pi,100);
>>plot(x,y)
```

For comparison, the exact values to four decimal places are given below.

```
x =
0
0.1000
0.2000
0.3000
0.4000
0.5000
0.6000
```

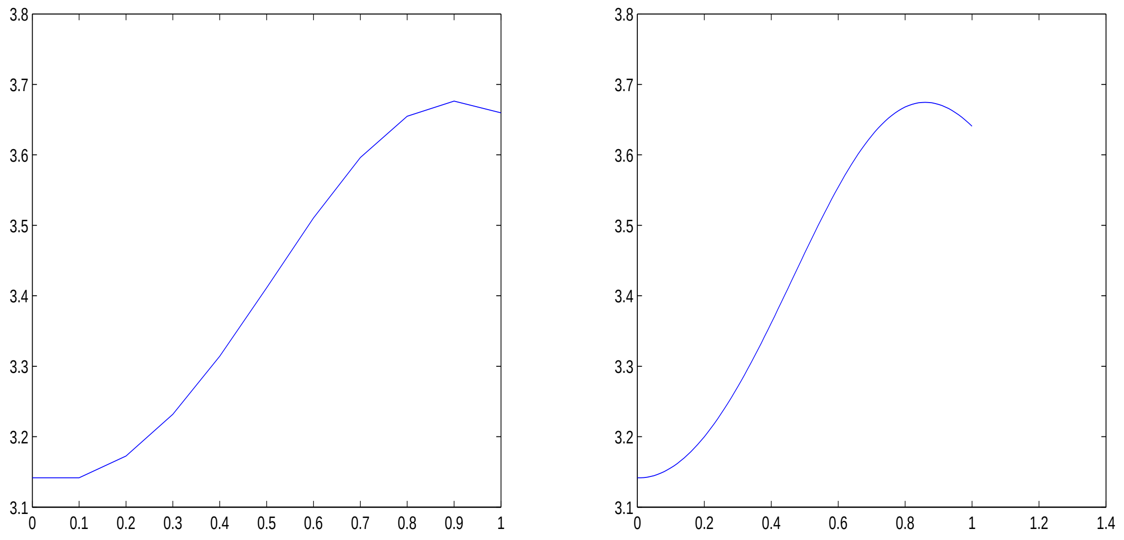


Figure 5: Euler approximations to the solution of $y' = \sin(x, y)$ with $\triangle x = .1$ (left) and $\triangle x = .01$ (right).

```
0.7000
0.8000
0.9000
1.0000
y =
3.1416
3.1572
3.2029
3.2750
3.3663
3.4656
3.5585
3.6299
3.6688
3.6708
3.6383
```

$\triangle$

5.2 Higher order Taylor Methods

The basic idea of Euler's method can be improved in a straightforward manner by using higher order Taylor polynomials. In deriving Euler's method, we approximated $y(x)$ with a first order polynomial. More generally, if we use a Taylor polynomial of order n we obtain the *Taylor method of order n*. In order to see how this is done, we will derive the Taylor method of order 2. (Euler's method is the Taylor method of order 1.)

Again letting $P = [x_0, x_1, ..., x_n]$ denote a partition of the interval $[x_0, x_n]$ on which we would like to solve (7), our starting point for the Taylor method of order 2 is to write down the Taylor polynomial of order 2 (with remainder) for $y(x_{k+1})$ about the point x_k. That is, according to Taylor's theorem,

$$y(x_{k+1}) = y(x_k) + y'(x_k)(x_{k+1} - x_k) + \frac{y''(x_k)}{2}(x_{k+1} - x_k)^2 + \frac{y'''(c)}{3!}(x_{k+1} - x_k)^3,$$

where $c \in (x_k, x_{k+1})$. As with Euler's method, we drop off the error term (which is now smaller), and our approximation is

$$y(x_{k+1}) \approx y(x_k) + y'(x_k)(x_{k+1} - x_k) + \frac{y''(x_k)}{2}(x_{k+1} - x_k)^2.$$

We already know from our derivation of Euler's method that $y'(x_k)$ can be replaced with $f(x_k, y(x_k))$. In addition to this, we now need an expression for $y''(x_k)$. We can obtain this by differentiating the original equation $y'(x) = f(x, y(x))$. That is,

$$y''(x) = \frac{d}{dx}y'(x) = \frac{d}{dx}f(x, y(x)) = \frac{\partial f}{\partial x}(x, y(x)) + \frac{\partial f}{\partial y}(x, y(x))\frac{dy}{dx},$$

where the last equality follows from a generalization of the chain rule to functions of two variables (see Section 10.5.1 of the course text). From this last expression, we see that

$$y''(x_k) = \frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))y'(x_k) = \frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))f(x_k, y(x_k)).$$

Replacing $y''(x_k)$ with the right-hand side of this last expression, and replacing $y'(x_k)$ with $f(x_k, y(x_k))$, we conclude

$$\begin{aligned} y(x_{k+1}) \approx\, & y(x_k) + f(x_k, y(x_k))(x_{k+1} - x_k) \\ & + \Big[\frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))f(x_k, y(x_k))\Big]\frac{(x_{k+1} - x_k)^2}{2}. \end{aligned}$$

If we take subintervals of equal width $\triangle x = (x_{k+1} - x_k)$, this becomes

$$\begin{aligned} y(x_{k+1}) \approx\, & y(x_k) + f(x_k, y(x_k))\triangle x \\ & + \Big[\frac{\partial f}{\partial x}(x_k, y(x_k)) + \frac{\partial f}{\partial y}(x_k, y(x_k))f(x_k, y(x_k))\Big]\frac{\triangle x^2}{2}. \end{aligned}$$

Example 5.2. Use the Taylor method of order 2 with $n = 10$ to approximate a solution to the differential equation

$$\frac{dy}{dx} = \sin(xy); \quad y(0) = \pi,$$

on the interval $[0, 1]$.

We will carry out the first few iterations by hand and leave the rest as a homework assignment. To begin, we observe that

$$\begin{aligned} f(x,y) &= \sin(xy) \\ \frac{\partial f}{\partial x}(x,y) &= y\cos(xy) \\ \frac{\partial f}{\partial y}(x,y) &= x\cos(xy). \end{aligned}$$

If we let y_k denote an approximation for $y(x_k)$, the Taylor method of order 2 becomes

$$\begin{aligned} y_{k+1} &= y_k + \sin(x_k y_k)(.1) \\ &\quad + \Big[y_k\cos(x_k y_k) + x_k\cos(x_k y_k)\sin(x_k y_k)\Big]\frac{(.1)^2}{2}. \end{aligned}$$

Beginning with the point $(x_0, y_0) = (0, \pi)$, we compute

$$y_1 = \pi + \pi(.005) = 3.1573,$$

which is closer to the correct value of 3.1572 than was the approximation of Euler's method. We can now use the point $(x_1, y_1) = (.1, 3.1573)$ to compute

$$\begin{aligned} y_2 &= 3.1573 + \sin(.1\cdot 3.1573)(.1) \\ &\quad + \Big[3.1573\cos(.1\cdot 3.1573) + .1\cos(.1\cdot 3.1573)\sin(.1\cdot 3.1573)\Big]\frac{(.1)^2}{2} \\ &= 3.2035, \end{aligned}$$

which again is closer to the correct value than was the y_2 approximation for Euler's method. $\triangle$

5.3 Assignments

1. [5 pts] Alter the MATLAB M-file *euler.m* so that it employs the Taylor method of order 2 rather than Euler's method. Use your file with $n = 10$ to approximate the solution to the differential equation

$$\frac{dy}{dx} = \sin(xy); \quad y(0) = \pi,$$

on the interval $[0, 1]$. Turn in both your M-file and a list of values approximating y at $0, .1, .2, ...$ **Hint**. Suppose you would like MATLAB to compute the partial derivative of $f(x, y) = \sin(xy)$ and designate the result as the inline function *fx*. Use

```
>>syms x y;
>>f=inline('sin(x*y)')
f =
Inline function:
f(x,y) = sin(x*y)
```

```
>>fx=diff(f(x,y),x)
fx =
cos(x*y)*y
>>fx=inline(char(fx))
fx =
Inline function:
fx(x,y) = cos(x*y)*y
```

2. [5 pts] Proceeding similarly as in Section 5.2, derive the Taylor method of order 3.

6 Linear Algebra in MATLAB

MATLAB's built-in solvers for systems of ODE return solutions as matrices, and so before considering these solvers we will briefly discuss MATLAB's built-in tools for matrix manipulation. As suggested by its name, MATLAB (MATrix LABoratory) was originally developed as a set of numerical algorithms for manipulating matrices.

6.1 Defining and Referring to a Matrix

First, we have already learned how to define $1 \times n$ matrices, which are simply row vectors. For example, we can define the 1×5 matrix

$$\vec{v} = [\ 0 \quad .2 \quad .4 \quad .6 \quad .8\ ]$$

with

```
>>v=[0 .2 .4 .6 .8]
v =
0 0.2000 0.4000 0.6000 0.8000
>>v(1)
ans =
0
>>v(4)
ans =
0.6000
```

We also see in this example that we can refer to the entries in v with $v(1)$, $v(2)$, etc.

In defining a more general matrix, we proceed similarly by defining a series of row vectors, ending each with a semicolon.

Example 6.1. Define the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

in MATLAB.

We accomplish this the following MATLAB code, in which we also refer to the entries a_{21} and a_{12}.

```
>>A=[1 2 3;4 5 6;7 8 9]
A =
1 2 3
4 5 6
7 8 9
>>A(1,2)
ans =
2
>>A(2,1)
ans =
4
```

In general, we refer to the entry a_{ij} with $A(i, j)$. We can refer to an entire column or an entire row by replacing the index for the row or column with a colon. In the following code, we refer to the first column of A and to the third row.

```
>>A(:,1)
ans =
1
4
7
>>A(3,:)
ans =
7 8 9
```

△

6.2 Matrix Operations and Arithmetic

MATLAB adds and multiplies matrices according to the standard rules.

Example 6.2. For the matrices

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 0 & 1 & 3 \\ 2 & 4 & 1 \\ 6 & 1 & 8 \end{bmatrix},$$

compute $A + B$, AB, and BA.

Assuming A has already been defined in Example 9.1, we use

```
>>B=[0 1 3;2 4 1;6 1 8]
B =
0 1 3
2 4 1
6 1 8
>>A+B
ans =
1 3 6
6 9 7
13 9 17
>>A*B
ans =
22 12 29
46 30 65
```

```
70 48 101
>>B*A
ans =
25 29 33
25 32 39
66 81 96
```

In this example, notice the difference between AB and BA. While matrix addition is commutative, matrix multiplication is not. △

It is straightforward using MATLAB to compute several quantities regarding matrices, including the determinant of a matrix, the tranpose of a matrix, powers of a matrix, the inverse of a matrix, and the eigenvalues of a matrix.

Example 6.3. Compute the determinant, transpose, 5th power, inverse, and eigenvalues of the matrix

$$B = \begin{bmatrix} 0 & 1 & 3 \\ 2 & 4 & 1 \\ 6 & 1 & 8 \end{bmatrix}.$$

Assuming this matrix has already been defined in Example 8.2, we use respectively

```
>>det(B)
ans =
-76
>>B'
ans =
0 2 6
1 4 1
3 1 8
B^5
ans =
17652 7341 27575
14682 6764 22609
55150 22609 86096
>>B^(-1)
ans =
-0.4079 0.0658 0.1447
0.1316 0.2368 -0.0789
0.2895 -0.0789 0.0263
>>inv(B)
ans =
-0.4079 0.0658 0.1447
0.1316 0.2368 -0.0789
0.2895 -0.0789 0.0263
>>eig(B)
ans =
-1.9716
```

```
10.1881
3.7835
```

Notice that the inverse of B can be computed in two different ways, either as B^{-1} or with the M-file *inv.m.* △

In addition to returning the eigenvalues of a matrix, the command *eig* will also return the associated eigenvectors. For a given square matrix A, the MATLAB command

$$[P, D] = \text{eig}(A)$$

will return a diagonal matrix D with the eigenvalues of A on the diagonal and a matrix P whose columns are made up of the eigenvectors of A. More precisely, the first column of P will be the eigenvector associated with d_{11}, the second column of P will be the eigenvector associated with d_{22} etc.

Example 6.4. Compute the eigenvalues and eigenvectors for the matrix B from Example 8.3.

We use

```
>>[P,D]=eig(B)
P =
-0.8480 0.2959 -0.0344
0.2020 0.2448 -0.9603
0.4900 0.9233 0.2767
>>D =
-1.9716 0 0
0 10.1881 0
0 0 3.7835
```

We see that the eigenvalue–eigenvector pairs of B are

$$-1.9716, \begin{bmatrix} -.8480 \\ .2020 \\ .4900 \end{bmatrix}; \quad 10.1881, \begin{bmatrix} .2959 \\ .2448 \\ .9233 \end{bmatrix}; \quad 3.7835, \begin{bmatrix} -.0344 \\ -.9603 \\ .2767 \end{bmatrix}.$$

Let's check that these are correct for the first one. We should have $B\vec{v}_1 = -1.9716\vec{v}_1$. We can check this in MATLAB with the calculations

```
>>B*P(:,1)
ans =
1.6720
-0.3982
-0.9661
>>D(1,1)*P(:,1)
ans =
1.6720
-0.3982
-0.9661
```

△

6.3 Solving Systems of Linear Equations

Example 6.5. Solve the linear system of equations

$$\begin{aligned} x_1 + 2x_2 + 7x_3 &= 1 \\ 2x_1 - x_2 &= 3 \\ 9x_1 + 5x_2 + 4x_3 &= 0. \end{aligned}$$

We begin by defining the coefficients on the left hand side as a matrix and the right hand side as a vector. If

$$A = \begin{bmatrix} 1 & 2 & 7 \\ 2 & -1 & 0 \\ 9 & 5 & 4 \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \text{and} \quad \vec{b} = \begin{bmatrix} 1 \\ 3 \\ 0 \end{bmatrix},$$

then in matrix form this equation is

$$A\vec{x} = \vec{b},$$

which is solved by

$$\vec{x} = A^{-1}\vec{b}.$$

In order to compute this in MATLAB, we use the following code.

```
>>A=[1 2 7;2 -1 0;9 5 4]
A =
1 2 7
2 -1 0
9 5 4
>>b=[1 3 0]'
b =
1
3
0
>>x=A^(-1)*b
x =
0.6814
-1.6372
0.5133
>>x=A\b
x =
0.6814
-1.6372
0.5133
```

Notice that MATLAB understands multiplication on the left by A^{-1}, and interprets division on the left by A the same way. △

If the system of equations we want to solve has no solution or an infinite number of solutions, then A will not be invertible and we won't be able to proceed as in Example 8.5.

In such cases (that is, when $\det A = 0$), we can use the command *rref*, which reduces a matrix by Gauss–Jordan elimination.

Example 6.6. Find all solutions to the system of equations

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 1 \\ x_1 + x_3 &= 2 \\ 2x_1 + x_2 + 3x_3 &= 3. \end{aligned}$$

In this case, we define the augmented matrix A, which includes the right-hand side. We have

```
>>A=[1 1 2 1;1 0 1 2;2 1 3 3]
A =
1 1 2 1
1 0 1 2
2 1 3 3
>>rref(A)
ans =
1 0 1 2
0 1 1 -1
0 0 0 0
```

Returning to equation form, we have

$$\begin{aligned} x_1 + x_3 &= 2 \\ x_2 + x_3 &= -1. \end{aligned}$$

We have an infinite number of solutions

$$S = \{(x_1, x_2, x_3) : x_3 \in \mathbb{R}, x_2 = -1 - x_3, x_1 = 2 - x_3\}.$$

△

Example 6.7. Find all solutions to the system of equations

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 1 \\ x_1 + x_3 &= 2 \\ 2x_1 + x_2 + 3x_3 &= 2. \end{aligned}$$

Proceeding as in Example 8.6, we use

```
>>A=[1 1 2 1;1 0 1 2;2 1 3 2]
A =
1 1 2 1
1 0 1 2
2 1 3 2
>>rref(A)
ans =
1 0 1 0
0 1 1 0
0 0 0 1
```

In this case, we see immediately that the third equation asserts $0 = 1$, which means there is no solution to this system. △

6.4 Assignments

1. [2 pts] For the matrix

$$A = \begin{bmatrix} 1 & 2 & 1 \\ -2 & 7 & 3 \\ 1 & 1 & 1 \end{bmatrix}$$

compute the determinant, transpose, 3rd power, inverse, eigenvalues and eigenvectors.

2. [2 pts] For matrix A from Problem 1, let P denote the matrix created from the eigenvectors of A and compute

$$P^{-1}AP.$$

Explain how the result of this calculation is related to the eigenvalues of A.

3. [2 pts] Find all solutions of the linear system of equations

$$\begin{aligned} x_1 + 2x_2 - x_3 + 7x_4 &= 2 \\ 2x_1 + x_2 + 9x_3 + 4x_4 &= 1 \\ 3x_1 + 11x_2 - x_3 + 7x_4 &= 0 \\ 4x_1 - 6x_2 - x_3 + 1x_4 &= -1. \end{aligned}$$

4. [2 pts] Find all solutions of the linear system of equations

$$\begin{aligned} -3x_1 - 2x_2 + 4x_3 &= 0 \\ 14x_1 + 8x_2 - 18x_3 &= 0 \\ 4x_1 + 2x_2 - 5x_3 &= 0. \end{aligned}$$

5. [2 pts] Find all solutions of the linear system of equations

$$\begin{aligned} x_1 + 6x_2 + 4x_3 &= 1 \\ 2x_1 + 4x_2 - 1x_3 &= 0 \\ -x_1 + 2x_2 + 5x_3 &= 0. \end{aligned}$$

7 Solving ODE Numerically in MATLAB

MATLAB has a number of built-in tools for numerically solving ordinary differential equations. We will focus on one of its most rudimentary solvers, *ode45*, which implements a version of the Runge–Kutta 4th order algorithm. (This is essentially the Taylor method of order 4, though implemented in an extremely clever way that avoids partial derivatives.) For a complete list of MATLAB's solvers, type *helpdesk* and then search for *nonlinear numerical methods.*

7.1 First Order Equations

Example 7.1. Numerically approximate the solution of the first order differential equation

$$\frac{dy}{dx} = xy^2 + y; \quad y(0) = 1,$$

on the interval $x \in [0, .5]$.

For any differential equation in the form $y' = f(x, y)$, we begin by defining the function $f(x, y)$. For single equations, we can define $f(x, y)$ as an inline function. Here,

```
>>f=inline('x*y^2+y')
f =
Inline function:
f(x,y) = x*y^2+y
```

The basic usage for MATLAB's solver *ode45* is

ode45(function,domain,initial condition).

That is, we use

```
>>[x,y]=ode45(f,[0 .5],1)
```

and MATLAB returns two column vectors, the first with values of x and the second with values of y. (The MATLAB output is fairly long, so I've omitted it here.) Since x and y are vectors with corresponding components, we can plot the values with

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

which creates Figure 6.

Choosing the partition. In approximating this solution, the algorithm *ode45* has selected a certain partition of the interval $[0, .5]$, and MATLAB has returned a value of y at each point in this partition. It is often the case in practice that we would like to specify the partition of values on which MATLAB returns an approximation. For example, we might only want to approximate $y(.1)$, $y(.2)$, ..., $y(.5)$. We can specify this by entering the vector of values $[0, .1, .2, .3, .4, .5]$ as the domain in *ode45.* That is, we use

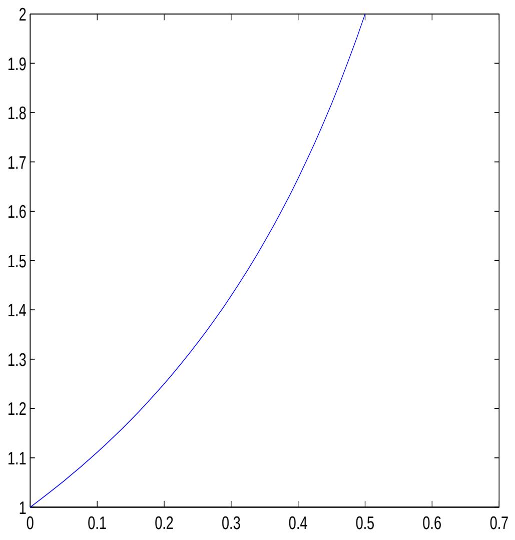


Figure 6: Plot of the solution to $y' = xy^2 + y$, with $y(0) = 1$.

```
>>xvalues=0:.1:.5
xvalues =
0 0.1000 0.2000 0.3000 0.4000 0.5000
>>[x,y]=ode45(f,xvalues,1)
x =
0
0.1000
0.2000
0.3000
0.4000
0.5000
y =
1.0000
1.1111
1.2500
1.4286
1.6667
2.0000
```

It is important to notice here that MATLAB continues to use roughly the same partition of values that it originally chose; the only thing that has changed is the values at which it is printing a solution. In this way, no accuracy is lost.

Options. Several options are available for MATLAB's *ode45* solver, giving the user limited control over the algorithm. Two important options are relative and absolute tolerance, respecively *RelTol* and *AbsTol* in MATLAB. At each step of the *ode45* algorithm, an error is approximated for that step[2]. If y_k is the approximation of $y(x_k)$ at step k, and e_k is the approximate error at this step, then MATLAB chooses its partition to ensure

$$e_k \leq \max(\text{RelTol} * y_k, \text{AbsTol}),$$

where the default values are RelTol = .001 and AbsTol = .000001. In the event that y_k becomes large, the value for *RelTol*y_k* will be large, and *RelTol* will need to be reduced. On the other hand, if y_k becomes small, *AbsTol* will be relatively large and will need to be reduced.

For the equation $y' = xy^2 + y$, with $y(0) = 1$, the values of y get quite large as x nears 1. In fact, with the default error tolerances, we find that the command

```
>>[x,y]=ode45(f,[0,1],1);
```

leads to an error message, caused by the fact that the values of y are getting too large as x nears 1. (Note at the top of the column vector for y that it is multipled by 10^{14}.) In order to fix this problem, we choose a smaller value for RelTol.

```
>>options=odeset('RelTol',1e-10);
>>[x,y]=ode45(f,[0,1],1,options);
>>max(y)
ans =
2.425060345544448e+07
```

In addition to employing the option command, I've computed the maximum value of $y(x)$ to show that it is indeed quite large, though not as large as suggested by the previous calculation. △

7.2 Systems of ODE

Solving a system of ODE in MATLAB is quite similar to solving a single equation, though since a system of equations cannot be defined as an inline function, we must define it as an M-file.

Example 7.2. Solve the Lotka–Volterra predator–prey system

$$\begin{aligned}\frac{dy_1}{dt} &= ay_1 - by_1y_2; \quad y_1(0) = y_1^0 \\ \frac{dy_2}{dt} &= -ry_2 + cy_1y_2; \quad y_2(0) = y_2^0,\end{aligned}$$

with $a = .5486$, $b = .0283$, $r = .8375$, $c = .0264$, and $y_1^0 = 30$, $y_2^0 = 4$. (These values correspond with data collected by the Hudson Bay Company between 1900 and 1920; see Sections 10 and 11 of these notes.)

In this case, we begin by writing the right hand side of this equation as the MATLAB M-file *lv.m:*

[2]In these notes I'm omitting any discussion of how one computes these approximate errors. The basic idea is to keep track of the Taylor remainder terms that we discarded in Section 7 in our derivation of the methods.

```
function yprime = lv(t,y)
%LV: Contains Lotka-Volterra equations
a = .5486;b = .0283;c = .0264;r = .8375;
yprime = [a*y(1)-b*y(1)*y(2);-r*y(2)+c*y(1)*y(2)];
```

We can now solve the equations over a 20 year period as follows:

```
>>[t,y]=ode45(@lv,[0 20],[30;4])
```

The output from this command consists of a column of times and a matrix of populations. The first column of the matrix corresponds with y_1 (prey in this case) and the second corresponds with y_2 (predators). Suppose we would like to plot the prey population as a function of time. In general, $y(n, m)$ corresponds with the entry of y at the intersection of row n and column m, so we can refer to the entire first column of y with $y(:, 1)$, where the colon means all entries. In this way, Figure 7 can be created with the command

```
>>plot(t,y(:,1))
```

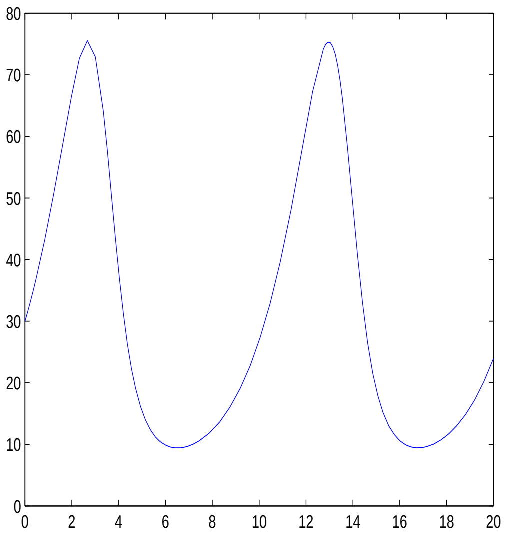


Figure 7: Prey populations as a function of time.

Similarly, we can plot the predator population along with the prey population with

```
>>plot(t,y(:,1),t,y(:,2),'–')
```

where the predator population has been dashed (see Figure 8).

Finally, we can plot an integral curve in the phase plane with

```
>>plot(y(:,1),y(:,2))
```

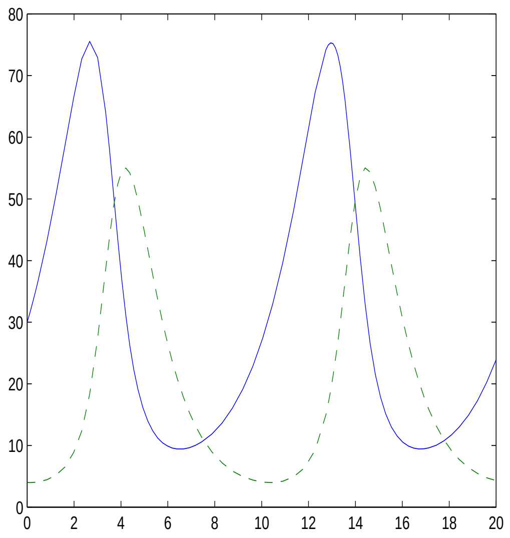


Figure 8: Plot of predator and prey populations for the Lotka–Volterra model.

which creates Figure 9. In this case, the integral curve is closed, and we see that the populations are cyclic. (Such an integral curve is called a *cycle*.)

Passing parameters. In analyzing systems of differential equations, we often want to experiment with parameter values. For example, we might want to solve the Lotka–Volterra system with different values of a, b, c, and r. In fact, we would often like a computer program to try different values of these parameters (typically while searching for a set that correspond with certain behavior), and so we require a method that doesn't involve manually altering our ODE file every time we want to change parameter values. We can accomplish this by passing the parameters through the function statement. In order to see how this works, we will alter *lv.m* so that the values a, b, c, and r are taken as input. We have

```
function yprime = lvparams(t,y,params)
%LVPARAMS: Contains Lotka-Volterra equations
a=params(1); b=params(2); c=params(3); r=params(4);
yprime = [a*y(1)-b*y(1)*y(2);-r*y(2)+c*y(1)*y(2)];
```

We can now send the parameters with *ode45*.

```
>>params = [.5486 .0283 .0264 .8375];
>>[t,y]=ode45(@lvparams,[0 20],[30;4],[],params);
```

MATLAB requires a statement in the position for options, so I have simply placed an empty set here, designated by two square brackets pushed together. △

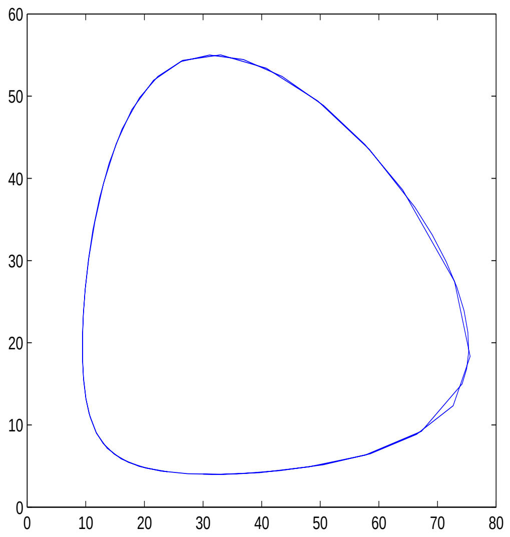


Figure 9: Integral curve for the Lotka–Volterra equation.

7.3 Higher Order Equations

In order to solve a higher order equation numerically in MATLAB we proceed by first writing the equation as a first order system and then proceeding as in the previous section.

Example 7.3. Solve the second order equation

$$y'' + x^2 y' - y = \sin x; \quad y(0) = 1, y'(0) = 3$$

on the interval $x \in [0, 5]$.

In order to write this equation as a first order system, we set $y_1(x) = y(x)$ and $y_2(x) = y'(x)$. In this way,

$$\begin{aligned} y_1' &= y_2; \quad y_1(0) = 1 \\ y_2' &= y_1 - x^2 y_2 + \sin x; \quad y_2(0) = 3, \end{aligned}$$

which is stored in the MATLAB M-file *secondode.m.*

```
function yprime = secondode(x,y);
%SECONDODE: Computes the derivatives of y_1 and y_2,
%as a colum vector
yprime = [y(2); y(1)-x^2*y(2)+sin(x)];
```

We can now proceed with the following MATLAB code, which produces Figure 10.

```
>>[x,y]=ode45(@secondode,[0,5],[1;3]);
>>plot(x,y(:,1))
```

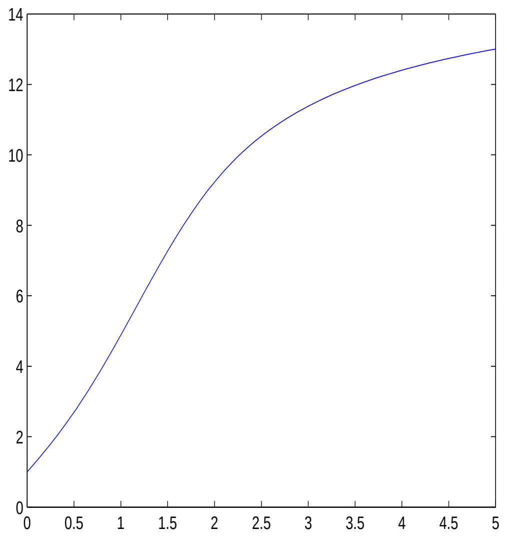


Figure 10: Plot of the solution to $y'' + x^2 y' - y = \sin x; \quad y(0) = 1, y'(0) = 3.$

7.4 Assignments

1. [2 pts] Numerically approximate the solution of the first order differential equation

$$\frac{dy}{dx} = \frac{x}{y} + y; \quad y(0) = 1,$$

on the interval $x \in [0, 1]$. Turn in a plot of your approximation.

2. [2 pts] Repeat Problem 1, proceeding so that MATLAB only returns values for $x_0 = 0$, $x_1 = .1$, $x_2 = .2$, ..., $x_{10} = 1$. Turn in a list of values $y(x_1)$, $y(x_2)$ etc.

3. [2 pts] The logistic equation

$$\frac{dp}{dt} = rp(1 - \frac{p}{K}); \quad p(0) = p_0,$$

is solved by

$$p_{\text{exact}}(t) = \frac{p_0 K}{p_0 + (K - p_0)e^{-rt}}.$$

For parameter values $r = .0215$, $K = 446.1825$, and for initial value $p_0 = 7.7498$ (consistent with the U. S. population in millions), solve the logistic equation numerically for $t \in [0, 200]$, and compute the error

$$E = \sum_{t=1}^{200} (p(t) - p_{\text{exact}}(t))^2.$$

4. [2 pts] Repeat Problem 3 with RelTol $= 10^{-10}$.

5. [2 pts] Numerically approximate the solution of the SIR epidemic model

$$\begin{aligned}\frac{dy_1}{dt} &= -by_1y_2; \quad y_1(0) = y_1^0\\ \frac{dy_2}{dt} &= by_1y_2 - ay_2; \quad y_2(0) = y_2^0,\end{aligned}$$

with parameter values $a = .4362$ and $b = .0022$ and initial values $y_1^0 = 762$ and $y_2^0 = 1$. Turn in a plot of susceptibles and infectives as a function of time for $t \in [0, 20]$. Also, plot y_2 as a function of y_1, and discuss how this plot corresponds with our stability analysis for this equation. Check that the maximum of this last plot is at $\frac{a}{b}$, and give values for S_f and S_R as defined in class.

8 Graphing Functions of Two Variables

In this section we will begin to get a feel for functions of two variables by plotting a number of functions with MATLAB. In the last subsection we will employ what we've learned to develop a geometric interpretation of partial derivatives.

8.1 Surface Plots

For a given function of two variables $f(x,y)$ we can associate to each pair (x,y) a third variable $z = f(x,y)$ and in this way create a collection of points (x,y,z) that describe a surface in three dimensions.

Example 8.1. Plot a graph of the function

$$f(x,y) = x^2 + 4y^2$$

for $x \in [-2,2]$ and $y \in [-1,1]$.

MATLAB provides a number of different tools for plotting surfaces, of which we will only consider *surf*. (Other options include *surfc*, *surfl*, *ezsurf, mesh*, *meshz, meshc*, *ezmesh.*) In principle, each tool works the same way: MATLAB takes a grid of points (x,y) in the specified domain, computes $z = f(x,y)$ at each of these points, and then plots the surface described by the resulting points (x,y,z). Given a vector of x values and a vector of y values, the command *meshgrid* creates such a mesh. In the following MATLAB code we will use the *surf* command to create Figure 11.

```
>>x=linspace(-2,2,50);
>>y=linspace(-1,1,50);
>>[x,y]=meshgrid(x,y);
>>z=x.^2+4*y.^2;
>>surf(x,y,z)
>>xlabel('x'); ylabel('y'); zlabel('z');
```

Here, the last command has been entered after the figure was minimized. Observe here that one tool in the graphics menu is a rotation option that allows us to look at the surface from several different viewpoints. Two different views are given in Figure 11. (These figures will appear in color in your MATLAB graphics window, and the different colors correspond with different levels of z.) △

Example 8.2. Plot the *ellipsoid* described by the equation

$$\frac{x^2}{4} + \frac{y^2}{16} + \frac{z^2}{9} = 1,$$

for appropriate values of x and y.

In this case the surface is described implicitly, though we can describe it explicitly as the two functions

$$z_- = -3\sqrt{1 - \frac{x^2}{4} - \frac{y^2}{9}}$$
$$z_+ = +3\sqrt{1 - \frac{x^2}{4} - \frac{y^2}{9}}.$$

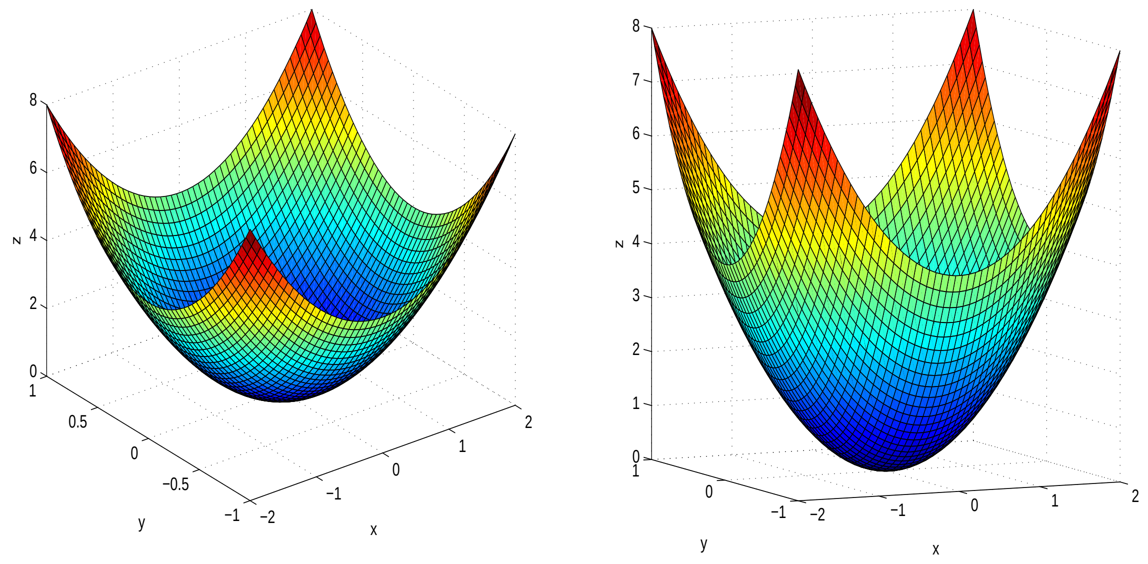


Figure 11: Graph of the paraboloid for $f(x, y) = x^2 + 4y^2$ (two views).

The difficulty we would have with proceeding as in the previous example is that the appropriate domain of x and y values is the ellipse described by

$$\frac{x^2}{4} + \frac{y^2}{16} \leq 1,$$

and *meshgrid* defines a rectangular domain. Fortunately, MATLAB has a built-in function *ellipsoid* that creates a mesh of points (x, y, z) an the surface of an ellipsoid described by the general equation

$$\frac{(x-a)^2}{A^2} + \frac{(y-b)^2}{B^2} + \frac{(z-c)^2}{C^2} = 1.$$

The syntax is

```
[x,y,z]=ellipsoid(a,b,c,A,B,C,n),
```

where n is an optional parameter ($n = 20$, by default) that sets the matrix of values describing each variable to be $(n + 1) \times (n + 1)$. Figure 12 with the following MATLAB code.

```
>>[x,y,z]=ellipsoid(0,0,0,2,4,3);
>>surf(x,y,z)
```

△

Example 8.3. Plot the *hyperboloid of one sheet* descibed by the equation

$$\frac{x^2}{4} - \frac{y^2}{16} + z^2 = 1.$$

As in Example 12.2, we run into a problem of specifying an appropriate domain of points (x, y), except that in this case MATLAB does not have (that I'm aware of) a built-in function

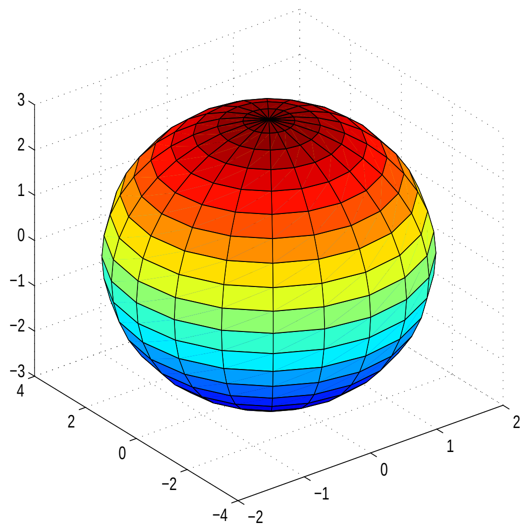


Figure 12: Ellipsoid described by $\frac{x^2}{4}+\frac{y^2}{16}+\frac{z^2}{9}=1$.

for graphing hyperbolas. Nonetheless, we can proceed by specifying the values x, y, and z parametrically. First, recall the hyperbolic trigonometric functions

$$\cosh s = \frac{e^s + e^{-s}}{2}$$
$$\sinh s = \frac{e^s - e^{-s}}{2}.$$

The following identity is readily verified: $\cosh^2 s - \sinh^2 s = 1$. If we additionally use the identity $\cos^2 t + \sin^2 t = 1$, then we find

$$\cosh^2(s)\cos^2(t) - \sinh^2(s) + \cosh^2(s)\sin^2(t) = 1.$$

What this means is that if we make the substitutions

$$x = 2\cosh(s)\cos(t)$$
$$y = 4\sinh(s)$$
$$z = \cosh(s)\sin(t),$$

then

$$\frac{x^2}{4} - \frac{y^2}{16} + z^2 = \frac{4\cosh^2(s)\cos^2(t)}{4} - \frac{16\sinh^2(s)}{16} + \cosh^2(s)\sin^2(t) = 1.$$

That is, these values of x, y, and z solve our equation. It is natural now to take $t \in [0, 2\pi]$ so that the cosine and sine functions will taken on all values between -1 and +1. The larger the interval we choose for s the larger the more of the manifold we'll plot. In this case, let's take $s \in [-1, 1]$. We can now create Figure 13 with the followoing MATLAB code.

```
>>s=linspace(-1,1,25);
>>t=linspace(0,2*pi,50);
>>[s,t]=meshgrid(s,t);
>>x=2*cosh(s).*cos(t);
>>y=4*sinh(s);
>>z=cosh(s).*sin(t);
>>surf(x,y,z)
```

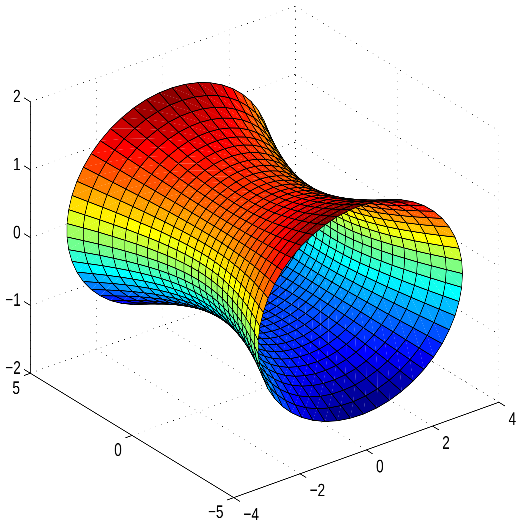


Figure 13: Plot of the surface for $\frac{x^2}{4} - \frac{y^2}{16} + z^2 = 1$.

Of course the ellipsoid plotted in Example 12.2 could similarly have been plotted with the parametrization

$$\begin{aligned} x &= 2\sin(s)\cos(t) \\ y &= 4\sin(s)\sin(t) \\ z &= 3\cos(s), \end{aligned}$$

and $s \in [0, \pi]$, $t \in [0, 2\pi]$. △

8.2 Geometric Interpretation of Partial Derivatives

For a function of one variable $f(x)$, the derivative at a value $x = a$ has a very simple geometric interpretation as the slope of the line that is tangent to the graph of $f(x)$ at the point $(a, f(a))$. In particular, if we write the equation for this tangent line in the point–slope form

$$y - y_0 = m(x - x_0),$$

then the slope is $m = f'(a)$ and the point of tangency is $(x_0, y_0) = (a, f(a))$, and so we have

$$y - f(a) = f'(a)(x - a).$$

In this subsection our goal will be to point out a similar geometric interpretation for functions of two variables. (Though we'll also say something about the case of three and more variables, the geometry is difficult to envision in those cases.)

When we consider the geometry of a function of two variables we should replace the concept of a tangent line with that of a tangent plane. As an extremely simple example to ground ourselves with, consider the parabaloid depicted in Figure 11, and the point at the origin $(0, 0, 0)$. The tangent plane to the surface at this point is the x-y plane. (The notion of precisely what we mean by a *tangent plane* will emerge from this discussion, but just so we're not chasing our tails, here's a version of it that does not involve derivatives: The tangent plane at a point P is defined as the plane such that for any sequence of points $\{P_n\}$ on the surface S such that $P_n \to P$ as $n \to \infty$, the angles between the lines PP_n and the plane approach 0.)

In order to determine an equation for the plane tangent to the graph of $f(x, y)$ at the point $(x_0, y_0, f(x_0, y_0))$, we first recall that just as every line can be written in the point–slope form $y - y_0 = m(x - x_0)$ every plane can be written in the form

$$z - z_0 = m_1(x - x_0) + m_2(y - y_0),$$

where (x_0, y_0, z_0) is a point on the plane. In order to find the value of m_1, we set $y = y_0$, so that z becomes a function of the single variable x:

$$z = f(x, y_0).$$

The line that is tangent to the graph of $f(x, y_0)$ at $x = x_0$ can be written in the point-slope form

$$z - z_0 = m_1(x - x_0),$$

where

$$m_1 = \frac{\partial f}{\partial x}(x_0, y_0).$$

Likewise, we find that

$$m_2 = \frac{\partial f}{\partial y}(x_0, y_0),$$

and so the equation for the tangent plane is

$$z - z_0 = \frac{\partial f}{\partial x}(x_0, y_0)(x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0)(y - y_0).$$

We conclude that $\frac{\partial f}{\partial x}(x_0, y_0)$ is a measure of the incline of the tangent plane in the x direction, while $\frac{\partial f}{\partial y}(x_0, y_0)$ is a measure of this incline in the y direction.

Example 8.4. Find the equation for the plane that is tangent to the paraboloid discussed in Example 12.1 at the point $(x_0, y_0) = (1, \frac{1}{2})$ and plot this tangent plane along with a surface plot of the function.

First, we observe that for $(x_0, y_0) = (1, \frac{1}{2})$, we have $z_0 = 1 + 1 = 2$. Additionally,

$$\frac{\partial f}{\partial x} = 2x \Rightarrow \frac{\partial f}{\partial x}(1, \frac{1}{2}) = 2$$
$$\frac{\partial f}{\partial y} = 8y \Rightarrow \frac{\partial f}{\partial y}(1, \frac{1}{2}) = 4.$$

The equation of the tangent plane is therefore

$$z - 2 = 2(x - 1) + 4(y - \frac{1}{2}),$$

or

$$z = 2x + 4y - 2.$$

We can now create Figure 14 with the following MATLAB code (an additional view has been added by a rotation in the graphics menu):

```
>>x=linspace(-2,2,50);
>>y=linspace(-1,1,50);
>>[x,y]=meshgrid(x,y);
>>z=x.^2+4*y.^2;
>>tplane=2*x+4*y-2;
>>surf(x,y,z)
>>hold on
>>surf(x,y,tplane)
```

$\triangle$

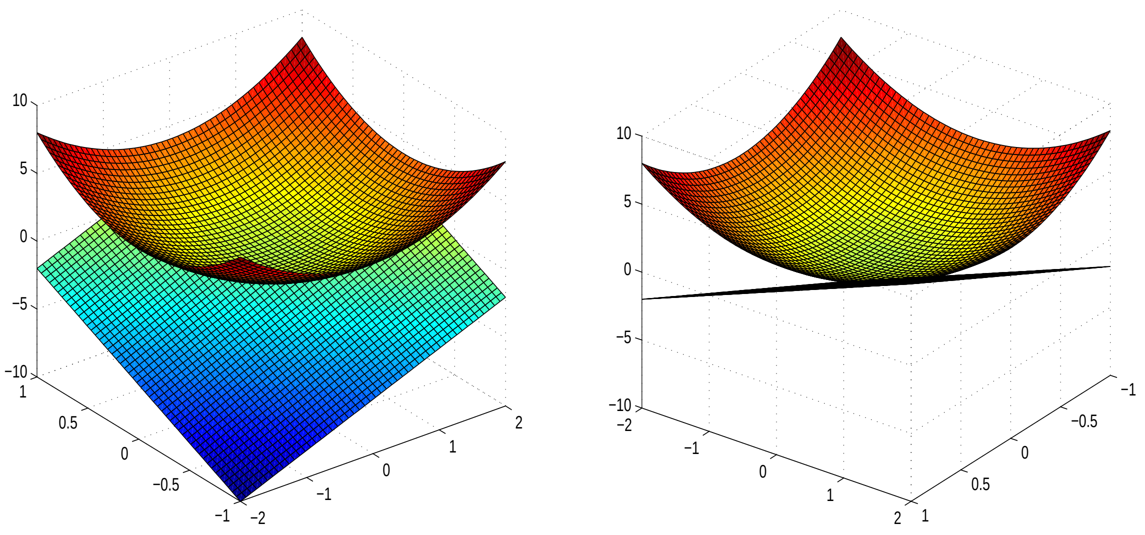


Figure 14: Graph of $f(x, y) = x^2 + 4y^2$ along with the tangent plane at $(1, \frac{1}{2})$ (two views).

8.3 Assignments

1. [2 pts] Create a surface plot for the function

$$f(x, y) = x^2 - 4y^2$$

for $x \in [-2, 2]$ and $y \in [-1, 1]$. Print two different views of your figure. (The result is a *hyperbolic paraboloid*, for which the point $(0, 0, 0)$ is a *saddle* point.)

2. [2 pts] Plot the ellipse described by the equation

$$\frac{x^2}{34.81} + \frac{y^2}{10.5625} + \frac{z^2}{10.5625} = 1.$$

(These are the dimensions in inches of a standard (American) football.)

3. [3 pts] Plot the surface described by the equation

$$\frac{x^2}{4} - \frac{y^2}{16} - z^2 = 1.$$

Hint. Observe the identity

$$\cosh^2(s) - \sinh^2(s)\cos^2(t) - \sinh^2(s)\sin^2(t) = 1.$$

This surface is called a *hyperboloid of two sheets,* so be sure your figure has two separate surfaces.

4. [3 pts] Along with your hyperbolic paraboloid from Problem 1, plot the plane that is tangent to your surface at the point $(x, y) = (1, \frac{1}{2})$.

9 Least Squares Regression

9.1 Data Fitting in MATLAB

One important application of maximization/minimization techniques is to the study of data fitting and parameter estimation.

Example 9.1. Suppose the Internet auctioneer, eBay, hires us to predict its net income for the year 2003, based on its net incomes for 2000, 2001, and 2002 (see Table 1).

Year	Net Income
2000	48.3 million
2001	90.4 million
2002	249.9 million

Table 1: Yearly net income for eBay.

We begin by simply plotting this data as a *scatterplot* of points. In MATLAB, we develop Figure 15 through the commands,

```
>>year=[0 1 2];
>>income=[48.3 90.4 249.9];
>>plot(year,income,'o')
>>axis([-.5 2.5 25 275])
```

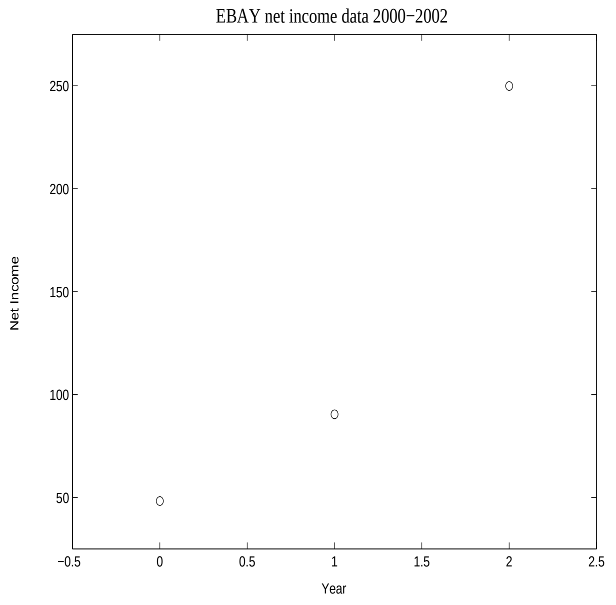


Figure 15: Net Income by year for eBay.

Our first approach toward predicting eBay's future profits might be to simply find a curve that best fits this data. The most common form of curve fitting is *linear least squares* *regression.* Before discussing the details of this method, we will get some idea of the goal of it by carrying out a calculation in MATLAB. At this point the only thing we need to understand regarding the underlying mathematics is that our goal is to find a polynomial that fits our data in some reasonable sense. The MATLAB command for polynomial fitting is *polyfit(x,y,n)*, where x and y are vectors and n is the order of the polynomial. For example, if we would like to fit the eBay data with a straight line (a polynomial of order 1), we have

```
>>polyfit(year,income,1)
ans =
100.8000 28.7333
```

Notice particularly that, left to right, MATLAB returns the coefficient of the highest power of x first, the second highest power of x second etc., continuing until the y-intercept is given last. Our model for the eBay data is

$$y = 100.8x + 28.7333,$$

where x denotes the year and y denotes the net income. Alternatively, for polynomial fitting up to order 10, MATLAB has the option of choosing it directly from the graphics menu. In the case of our eBay data, while Figure 15 is displayed in MATLAB, we choose **Tools, Basic Fitting**. A new window opens and offers a number of fitting options. We can easily experiment by choosing the **linear** option and then the **quadratic** option, and comparing. (Since we only have three data points in this example, the quadratic fit necessarily passes through all three. This, of course, does not mean that the quadratic fit is best, only that we need more data.) For this small a set of data, the linear fit is safest, so select that one and click on the black arrow at the bottom right corner of the menu. Checking that the fit given in the new window is linear, select the option **Save to Workspace**. MATLAB saves the polynomial fit as a *structure*, which is a MATLAB array variable that can hold data of varying types; for example, a string as its first element and a digit as its second and so on. The elements of a structure can be accessed through the notation *structurename.structureelement.* Here, the default structure name is *fit*, and the first element is *type.* The element *fit.type* contains a string describing the structure. The second element of the structure is *fit.coeff*, which contains the polynomial coefficients of our fit. Finally, we can make a prediction with the MATLAB command *polyval*,

```
>>polyval(fit.coeff,3)
```

for which we obtain the prediction 331.1333. Finally, we mention that MATLAB refers to the error for its fit as the *norm of residuals*, which is precisely the square root of the error E that we will define below. △

9.2 Polynomial Regression

In this section, we will develop the technique of least squares regression. To this end, the first thing we need is to specify what we mean by "best fit" in this context. Consider the three points of data given in Example 13.1, and suppose we would like to draw a line through

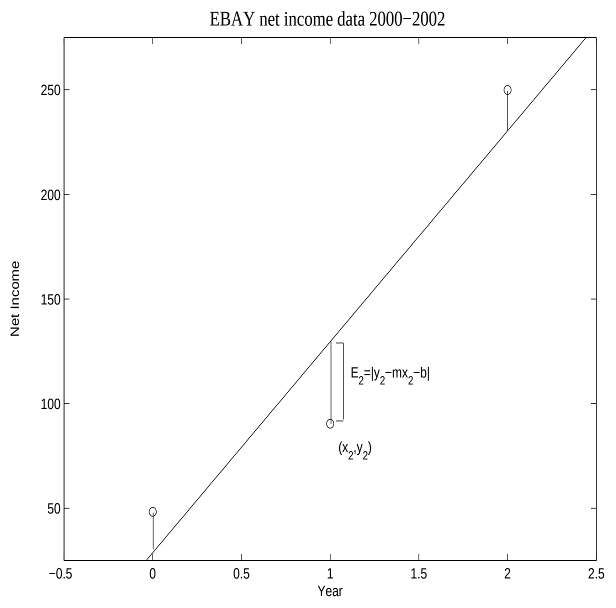


Figure 16: Least squares vertical distances.

these points in such a way that the distance between the points and the line is minimized (see Figure 16).

Labeling these three points (x_1, y_1), (x_2, y_2), and (x_3, y_3), we observe that the vertical distance between the line and the point (x_2, y_2) is given by the error $E_2 = |y_2 - mx_2 - b|$. The idea behind the least squares method is to sum these vertical distances and minimize the total error. In practice, we square the errors both to keep them positive and to avoid possible difficulty with differentiation (recall that absolute values can be subtle to differentiate), which will be required for minimization. Our total least squares error becomes

$$E(m,b) = \sum_{k=1}^{n} (y_k - mx_k - b)^2.$$

In our example, $n = 3$, though the method remains valid for any number of data points.

We note here that in lieu of these vertical distances, we could also use horizontal distances between the points and the line or direct distances (the shortest distances between the points and the line). While either of these methods could be carried out in the case of a line, they both become considerably more complicated in the case of more general cures. In the case of a parabola, for example, a point would have two different horizontal distances from the curve, and while it could only have one shortest distance to the curve, computing that distance would be a fairly complicted problem in its own right.

Returning to our example, our goal now is to find values of m and b that minimize the error function $E(m, b)$. In order to maximize or minimize a function of multiple variables, we compute the partial derivative with respect to each variable and set them equal to zero.

Here, we compute

$$\begin{aligned}\frac{\partial}{\partial m}E(m,b) &= 0\\ \frac{\partial}{\partial b}E(m,b) &= 0.\end{aligned}$$

We have, then,

$$\begin{aligned}\frac{\partial}{\partial m}E(m,b) &= -2\sum_{k=1}^{n} x_k(y_k - mx_k - b) = 0,\\ \frac{\partial}{\partial b}E(m,b) &= -2\sum_{k=1}^{n}(y_k - mx_k - b) = 0,\end{aligned}$$

which we can solve as a linear system of two equations for the two unknowns m and b. Rearranging terms and dividing by 2, we have

$$\begin{aligned}m\sum_{k=1}^{n} x_k^2 + b\sum_{k=1}^{n} x_k &= \sum_{k=1}^{n} x_k y_k,\\ m\sum_{k=1}^{n} x_k + b\sum_{k=1}^{n} 1 &= \sum_{k=1}^{n} y_k.\end{aligned} \tag{10}$$

Observing that $\sum_{k=1}^{n} 1 = n$, we multiply the second equation by $\frac{1}{n}\sum_{k=1}^{n} x_k$ and subtract it from the first to get the relation,

$$m\Big(\sum_{k=1}^{n} x_k^2 - \frac{1}{n}(\sum_{k=1}^{n} x_k)^2\Big) = \sum_{k=1}^{n} x_k y_k - \frac{1}{n}(\sum_{k=1}^{n} x_k)(\sum_{k=1}^{n} y_k),$$

or

$$m = \frac{\sum_{k=1}^{n} x_k y_k - \frac{1}{n}(\sum_{k=1}^{n} x_k)(\sum_{k=1}^{n} y_k)}{\sum_{k=1}^{n} x_k^2 - \frac{1}{n}(\sum_{k=1}^{n} x_k)^2}.$$

Finally, substituting m into equation (10), we have

$$\begin{aligned}b &= \frac{1}{n}\sum_{k=1}^{n} y_k - (\sum_{k=1}^{n} x_k)\frac{\sum_{k=1}^{n} x_k y_k - \frac{1}{n}(\sum_{k=1}^{n} x_k)(\sum_{k=1}^{n} y_k)}{n\sum_{k=1}^{n} x_k^2 - (\sum_{k=1}^{n} x_k)^2}\\ &= \frac{(\sum_{k=1}^{n} y_k)(\sum_{k=1}^{n} x_k^2) - (\sum_{k=1}^{n} x_k)(\sum_{k=1}^{n} x_k y_k)}{n\sum_{k=1}^{n} x_k^2 - (\sum_{k=1}^{n} x_k)^2}.\end{aligned}$$

We can verify that these values for m and b do indeed constitute a minimum by observing that by continuity there must be at least one local minimum for E, and that since m and b are uniquely determined this must be it. Alternatively, we have the following theorem.

Theorem 9.1. *Suppose $f(x,y)$ is differentiable in both independent variables on some domain $D \subset \mathbb{R}^2$ and that (a,b) is an interior point of D for which*

$$f_x(a,b) = f_y(a,b) = 0.$$

If moreover f_{xx}, f_{xy}, f_{yx}, and f_{yy} exist at the point (a,b), then the nature of (a,b) can be determined from

$$D = f_{xx}(a,b)f_{yy}(a,b) - f_{xy}(a,b)^2$$

as follows:

1. *If $D > 0$ and $f_{xx}(a,b) > 0$, then f has a relative minimum at (a,b);*
2. *If $D > 0$ and $f_{xx}(a,b) < 0$, then f has a relative maximum at (a,b);*
3. *If $D < 0$, then f has neither a maximum nor a minimum at (a,b);*
4. *If $D = 0$, further analysis is required, and any of 1–3 remain possible.*

Notice particularly that in the event that $D > 0$, this looks very much like the related second-derivative test for functions of a single independent variable.

Observe finally that we can proceed similarly polynomials of higher order. For example, for second order polynomials with general form $y = a_0 + a_1x + a_2x^2$, our error becomes

$$E(a_0, a_1, a_2) = \sum_{k=1}^{n}(y_k - a_0 - a_1x_k - a_2x_k^2)^2.$$

In this case, we must compute a partial derivative of E with respect to each of three parameters, and consequently (upon differentiation) solve three linear equations for the three unknowns.

We conclude this section with an example in which the best fit polynomial is quadratic.

Example 9.2. (Crime and Unemployment.) Table 2 contains data relating crime rate per 100,000 citizens to percent unemployment during the years 1994–2000. In this example, we will find a polynomial that describes crime rate as a function of percent unemployment.

Year	Crime Rate	Percent Unemployment
1994	5,373.5	6.1%
1995	5,277.6	5.6%
1996	5,086.6	5.4%
1997	4,922.7	4.9%
1998	4,619.3	4.5%
1999	4,266.8	4.2%
2000	4,124.8	4.0%

Table 2: Crime rate and unemployment data.

We begin by plotting these points with crime rates on the y-axis and percent unemployment on the x-axis. The following MATLAB code creates Figure 17.

```
>>x=[6.1 5.6 5.4 4.9 4.5 4.2 4.0];
>>y=[5373.5 5277.6 5086.6 4922.7 4619.3 4266.8 4124.8];
>>plot(x,y,'o')
```

Proceeding from the **Tools, Data Fitting** option in the graphics menu (as in Example 13.1), we try both the linear and the quadratic fit and find that the quadratic fit is much better (see Figure 18). Though we haven't yet quantitatively developed the socioeconomic reasons for this particular relation,[3] we do have a genuinely predictive model. For example,

[3]And don't worry, we're not going to.

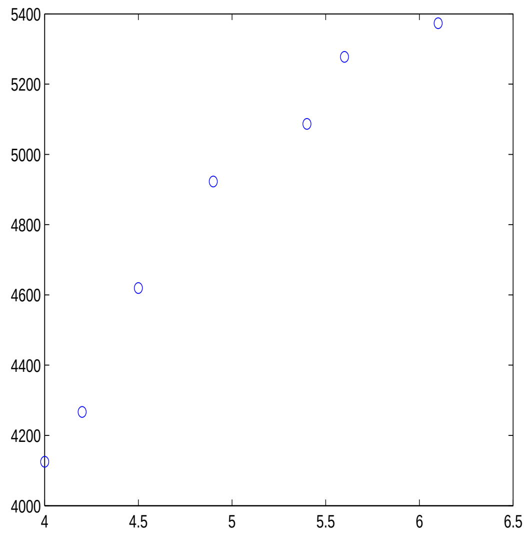


Figure 17: Scatterplot of crime rate versus unemployment.

the percent unemployment in 2001 was 4.8%, and so if we save our quadratic fit to the workplace as *fit2*, we can predict that the crime rate was 4,817.8 as follows[4]:

```
>>polyval(fit2.coeff,4.8)
ans =
4.8178e+03
```

△

9.3 Regression with more general functions

Example 9.3. Let's consider population growth in the United States, beginning with the first government census in 1790 (see Table 3).

Year	1790	1800	1810	1820	1830	1840	1850	1860	1870	1880	1890	1900
Pop	3.93	5.31	7.24	9.64	12.87	17.07	23.19	31.44	39.82	50.16	62.95	75.99

Year	1910	1920	1930	1940	1950	1960	1970	1980	1990	2000
Pop	91.97	105.71	122.78	131.67	151.33	179.32	203.21	226.50	249.63	281.42

Table 3: Population data for the United States, 1790–2000, measured in millions.

If we let $p(t)$ represent the population at time t, the *logistic* model of population growth is given by

$$\frac{dp}{dt} = rp(1 - \frac{p}{K}); \quad p(0) = p_0 \tag{11}$$

[4]As it turns out, this isn't a very good prediction, because there was a lag between the increase in percent employment (from 4% in 2000 to 4.8% in 2001) and the increase in crime rate, which only increased from 4124.8 in 2000 to 4,160.5 in 2001.

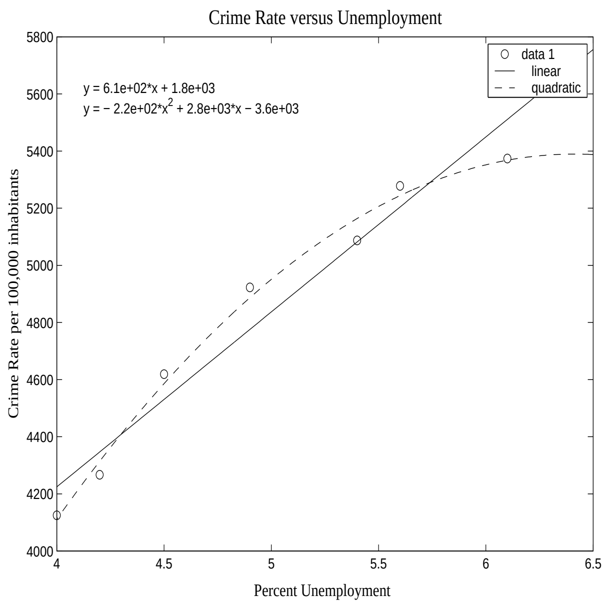


Figure 18: Best fit curves for crime–unemployment data.

where p_0 represents the initial population of the inhabitants, r is called the "growth rate" of the population, and K is called the "carrying capacity." Observe that while the rate at which the population grows is assumed to be proportional to the size of the population, the population is assumed to have a maximum possible number of inhabitants, K. (If $p(t)$ ever grows larger than K, then $\frac{dp}{dt}$ will become negative and the population will decline.) Equation (11) can be solved by separation of variables and partial fractions, and we find

$$p(t) = \frac{p_0 K}{(K - p_0)e^{-rt} + p_0}. \tag{12}$$

Though we will take year 0 to be 1790, we will assume the estimate that year was fairly crude and obtain a value of p_0 by fitting the entirety of the data. In this way, we have three parameters to contend with, and carrying out the full regression analysis would be tedious.

The first step in finding values for our parameters with MATLAB consists of writing our function $p(t)$ as an M-file, with our three parameters p_0, r, and K stored as a parameter vector $p = (p(1), p(2), p(3))$:

```
function P = logistic(p,t);
%LOGISTIC: MATLAB function file that takes
%time t, growth rate r (p(1)),
%carrying capacity K (p(2)),
%and initial population P0 (p(3)), and returns
%the population at time t.
P = p(2).*p(3)./((p(2)-p(3)).*exp(-p(1).*t)+p(3));
```

MATLAB's function *lsqcurvefit()* can now be employed at the Command Window, as follows.

```
>>decades=0:10:210;
>>pops=[3.93 5.31 7.24 9.64 12.87 17.07 23.19 31.44 39.82 50.16 62.95 75.99...
91.97 105.71 122.78 131.67 151.33 179.32 203.21 226.5 249.63 281.42];
>>p0 = [.01 1000 3.93];
>>[p,error]=lsqcurvefit(@logistic,p0,decades,pops)
Maximum number of function evaluations exceeded;
increase options.MaxFunEvals
p =
0.0206 485.2402 8.4645
error =
464.1480
>>sqrt(error)
ans =
21.5441
```

After defining the data, we have entered our initial guess as to what the parameters should be, the vector p_0. (Keep in mind that MATLAB is using a routine similar to *fzero()* to solve the system of nonlinear algebraic equations, and typically can only find roots reasonably near your guesses.) In this case, we have guessed a small value of r corresponding very roughly to the fact that we live 70 years and have on average (counting men and women) 1.1 children per person[5] ($r \cong 1/70$), a population carrying capacity of 1 billion, and an initial population equal to the census data. Finally, we use *lsqcurvefit()*, entering respectively our function file, our initial parameter guesses, and our data. The function *lsqcurvefit()* renders two outputs: our parameters and a sum of squared errors, which we have called *error*. Though the error looks enormous, keep in mind that this is a sum of all errors squared,

$$\text{error} = \sum_{\text{decades}} (\text{pops(decade)-modelpops(decade)})^2.$$

A more reasonable measure of error is the square root of this, from which we see that over 22 decades our model is only off by around 21.54 million people. In this case, MATLAB has not closed its iteration completely, and returns the error message, *Maximum number of function evaluations exceeded.* In order to avoid this, let's refine our code by increasing the number of function evaluations MATLAB should carry out. We have

```
>>lowerbound = [0 0 0];
>>options = optimset('MaxFunEvals',10000);
>>[p,error]=lsqcurvefit(@logistic,p0,decades,pops,lowerbound,[],options)
Optimization terminated successfully:
Relative function value changing by less than OPTIONS.TolFun
p =
0.0215 446.1825 7.7498
error =
440.9991
```

[5]According to census 2000.

```
>>sqrt(error)
ans =
21.0000
>>modelpops=logistic(p,decades);
>>plot(decades,pops,'o',decades,modelpops)
```

In the first line of this code, simply for the sake of example, we've included a lower bound of 0 on each parameter, while in the next line, we've used the command *optimset*, which gives us control over the maximum number of iterations *lsqcurvefit()* will do. In this case, we've set the maximum number of iterations to 10,000, which turns out to be sufficient, at least for meeting MATLAB's internal convergence requirements. In our new *lsqcurvefit()* command, the vector *lowerbound* is added at the end of the required inputs, followed by square brackets, [], which signify that we are not requiring an upper bound in this case. Finally, the options defined in the previous line are incorporated as the final input. (The MATLAB option *OPTIONS.TolFun* is a built-in tolerance value and can be reduced by the *optimset* command, but the default value is typically small enough.)

We see from MATLAB's output that our error has decreased slightly. For the parameters, we observe that r has remained small (roughly 1/50), our carrying capacity is 446 million people, and our initial population is 7.75 million. In the last two lines of code, we have created Figure 19, in which our model is compared directly with our data. △

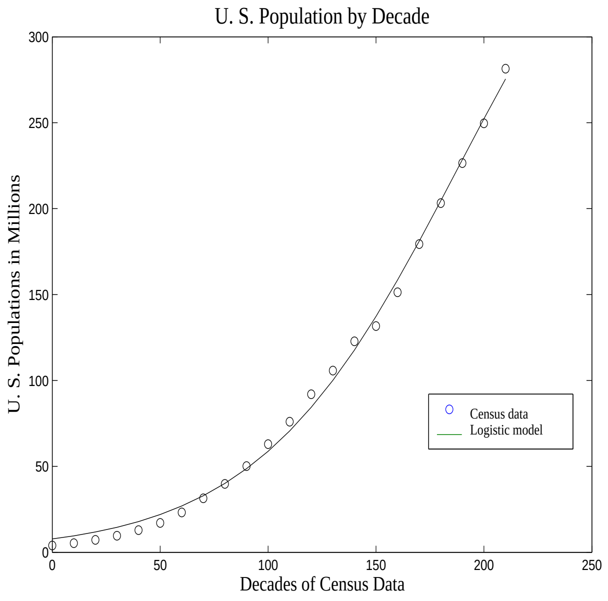


Figure 19: U.S. Census data and logistic model approximation.

9.4 Multivariate Regression

Often, the phenomenon we would like to model depends on more than one independent variable. (Keep in mind the following distinction: While the model in Example 13.3 depended on three parameters, it depended on only a single independent variable, t.)

Example 9.4. Film production companies such as Paramount Studios and MGM employ various techniques to predict movie ticket sales. In this example, we will consider the problem of predicting final sales based on the film's first weekend. The first difficulty we encounter is that first weekend sales often depend more on hype than quality. For example, *Silence of the Lambs* and *Dude, Where's My Car?* had surprisingly similar first-weekend sales: \$13,766,814 and \$13,845,914 respectively.[6] (*Dude* did a little better.) Their final sales weren't so close: \$130,726,716 and \$46,729,374, again respectively. Somehow, our model has to *numerically* distinguish between movies like *Silence of the Lambs* and *Dude, Where's My Car?* Probably the easiest way to do this is by considering a second variable, the movie's rating. First weekend sales, final sales and TV Guide ratings are listed for ten movies in Table 4.[7]

Movie	First Weekend Sales	Final Sales	Rating
Dude, Where's My Car?	13,845,914	46,729,374	1.5
Silence of the Lambs	13,766,814	130,726,716	4.5
We Were Soldiers	20,212,543	78,120,196	2.5
Ace Ventura	12,115,105	72,217,396	3.0
Rocky V	14,073,170	40,113,407	3.5
A.I.	29,352,630	78,579,202	3.0
Moulin Rouge	13,718,306	57,386,369	2.5
A Beautiful Mind	16,565,820	170,708,996	3.0
The Wedding Singer	18,865,080	80,245,725	3.0
Zoolander	15,525,043	45,172,250	2.5

Table 4: Movie Sales and Ratings.

Letting S represent first weekend sales, F represent final sales, and R represent ratings, our model will take the form

$$F = a_0 + a_1 S + a_2 R,$$

where a_0, a_1, and a_2 are parameters to be determined; that is, we will use a linear two-variable polynomial. For each set of data points from Table 4 (S_k, R_k, F_k) $(k = 1, ..., 10)$ we have an equation

$$F_k = a_0 + a_1 S_k + a_2 R_k.$$

Combining the ten equations (one for each k) into matrix form, we have

$$\begin{pmatrix} 1 & S_1 & R_1 \\ 1 & S_2 & R_2 \\ \vdots & \vdots & \vdots \\ 1 & S_{10} & R_{10} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_{10} \end{pmatrix}, \tag{13}$$

[6] All sales data for this example were obtained from http://www.boxofficeguru.com. They represent domestic (U. S.) sales.

[7] TV Guide's ratings were easy to find, so I've used them for the example, but they're actually pretty lousy. FYI.

or $Ma = F$, where

$$M = \begin{pmatrix} 1 & S_1 & R_1 \\ 1 & S_2 & R_2 \\ \vdots & \vdots & \vdots \\ 1 & S_{10} & R_{10} \end{pmatrix}, \qquad a = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}, \qquad \text{and} \qquad F = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_{10} \end{pmatrix}.$$

In general, equation (13) is over-determined—ten equations and only three unknowns. In the case of over-determined systems, MATLAB computes $a = M^{-1}F$ by linear least squares regression, as above, starting with the error $E = \sum_{k=1}^{10}(F_k - a_0 - a_1S_k - a_2R_k)^2$. Hence, to carry out regression in this case on MATLAB, we use the following commands:

```
>>S=[13.8 13.8 20.2 12.1 14.1 29.4 13.7 16.6 18.9 15.5];
>>F=[46.7 130.7 78.1 72.2 40.1 78.6 57.4 170.7 80.2 45.2];
>>R=[1.5 4.5 2.5 3.0 3.5 3.0 2.5 3.0 3.0 2.5];
>>M=[ones(size(S))' S' R'];
>>a=M\F'
a =
-6.6986
0.8005
25.2523
```

Notice that a prime (') has been set after each vector in the definition of M to change the row vectors into column vectors. (Alternatively, we could have typed a semicolon after each entry in each vector.) MATLAB's notation for $M^{-1}F$ is $M\backslash F$, which was one of MATLAB's earliest conventions, from the days when it was almost purely meant as a tool for matrix manipulations. Note finally that in the event that the number of equations is the same as the number of variables, MATLAB solves for the variables exactly (assuming the matrix is invertible).

At his point, we are in a position to make a prediction. In 2003, the Affleck–Lopez debacle *Gigli* opened with an initial weekend box office of \$3,753,518, and a *TV Guide* rating of 1.0 stars. Following the code above, we can predict the final box office for *Gigli* with the code:

```
>>a(1)+a(2)*3.8+a(3)*1
ans =
21.5957
```

The actual final sales of *Gigli* were \$6,087,542, wherein we conclude that our model is not quite sufficient for predicting that level of disaster.

Though scatterplots can be difficult to read in three dimensions, they are often useful to look at. In this case, we simply type *scatter3(S,R,F)* at the MATLAB Command Window prompt to obtain Figure 20.

While in the case of a single variable, regression found the best-fit line, for two variables it finds the best-fit plane. Employing the following MATLAB code, we draw the best fit plane that arose from our analysis (see Figure 21).

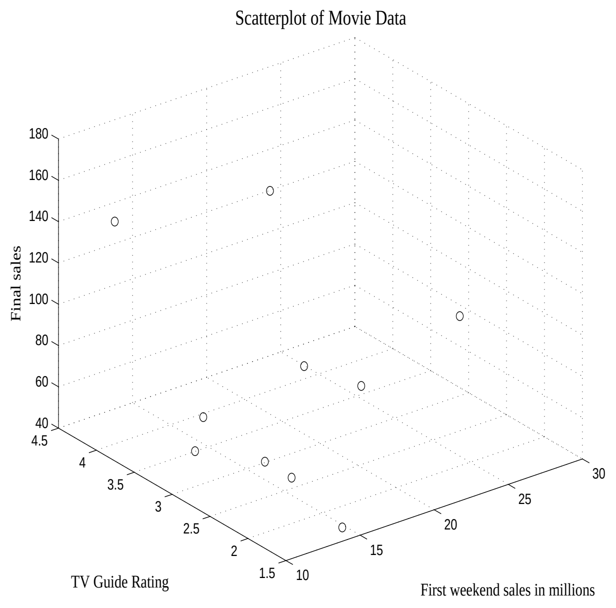


Figure 20: Scatterplot for movie sales data.

```
>>hold on
>>x=10:1:30;
>>y=1:.5:5;
>>[X,Y]=meshgrid(x,y);
>>Z=a(1)+a(2)*X+a(3)*Y;
>>surf(X,Y,Z)
```

9.5 Assignments

1. [3 pts] The terminology *regression* was coined by the British anthropologist Francis Galton (1822–1911), who used it in describing studies he was conducting on the correlation between adult height and offspring height. Though his first work in this regard involved sweet peas (1877), we'll discuss it in terms of the more famous analysis of human height (1885). In order to view child height as a function of parent height, Galton defined the *midheight* as

$$\text{Midheight} = \frac{\text{Father's height } + 1.08 \text{ Mother's height}}{2}.$$

Given the data in Table 5, find the linear regression line that describes Son's height as a function of midheight.

You might expect the best fit line to be roughly $y = x$, but what you will find is that the slope is less than 1. What this means is that if the midheight is larger than average, then the son's height will tend to be slightly smaller than the midheight, while if the midheight is smaller than average the son's height will be slightly larger than the midheight. Galton referred to this as "regression toward the mean." (Though Galton's calculation wasn't quite the same one we're doing.)

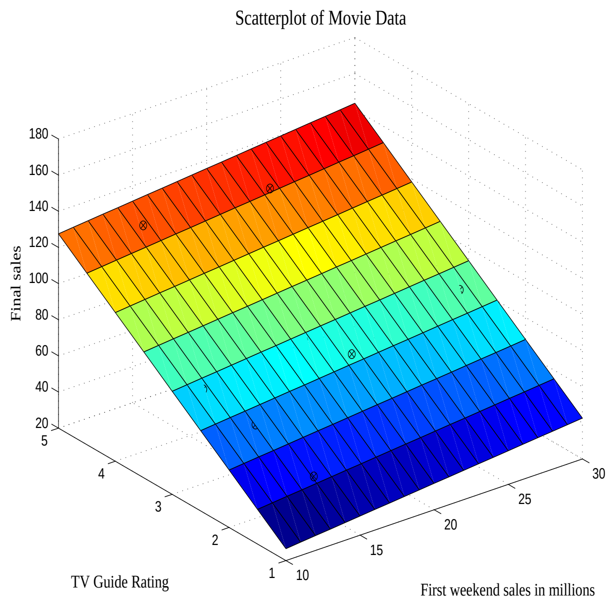


Figure 21: Movie sales data along with best-fit plane.

2. [3 pts] The *Gompertz* model for population growth is described by the equation

$$\frac{dp}{dt} = -rp\ln(\frac{p}{K}); \quad p(0) = p_0,$$

which (as we will show next semester) can be solved by the function

$$p(t) = K(\frac{p_0}{K})^{e^{-rt}}.$$

Using the U.S. population data from Example 13.3, estimate values for r, K, and p_0, and plot your model along with the data.

3. [4 pts] An alternative approach to the fit we carried out in Problem 1 would be a multivariate fit with

$$C = a_0 + a_1M + a_2F,$$

where C denotes child's height, M denotes mother's height, and F denotes father's height. Find values for a_0, a_1, and a_2 for this fit. According to this model, which height is more significant for a son's height, the mother's or the father's.

Mother's height	Father's height	Son's height
65	67	68
60	70	67
70	72	74
62	70	69
63	67	68
64	71	70
64	69	69
61	68	67
65	70	70
66	73	70

Table 5: Child and Parent Heights.

10 Estimating Parameters in Discrete Time Models

In many cases, we would like to find parameter values for a discrete time model even when we cannot write down an explicit solution to the model. In such cases we can still use least squares regression if we solve the equation numerically each time we need to evaluate it. While computationally expensive (i.e., time consuming), this process can give very good results.

Example 10.1. Given the U. S. population data in Table 3 from Example 13.3, find values for N_0, R, and K in the Beverton–Holt population model

$$N_{t+1} = \frac{RN_t}{1+\frac{R-1}{K}N_t}; \quad N_0 = N_0. \tag{14}$$

Your unit of time should be 1 year.

Our approach will be to apply least squares regression in the following way. We will solve (14) numerically, starting with the year 1790, and we will compare the value each ten years with the data. That is, we will compute the error function

$$E(N_0, R, K) = \sum_{\text{decades}} (\text{Pop (decade)} - N(\text{decade}))^2.$$

We will then find values of N_0, R, and K that minimize E. (Observe that we are treating the initial population as a parameter to be estimated, because we assume the first census probably wasn't particularly accurate.) In principle, we are finding the values of these parameters so that

$$\frac{\partial E}{\partial N_0} = 0$$
$$\frac{\partial E}{\partial R} = 0$$
$$\frac{\partial E}{\partial K} = 0.$$

In practice, we will carry out this minimization with MATLAB. First, for convenience, we write a MATLAB script M-file *pop.m* that defines the data for our problem.

```
%POP: Defines decades and U.S. populations.
decades=0:10:210;
pops=[3.93 5.31 7.24 9.64 12.87 17.07 23.19 31.44 39.82 50.16 62.95 75.99...
91.97 105.71 122.78 131.67 151.33 179.32 203.21 226.5 249.63 281.42];
```

Here, we are taking 1790 as year 0. Next, we write a MATLAB function M-file *bherror.m* that takes as input three values N_0, R, and K and returns a value for our error E.

```
function totalerror = bherror(p)
%p(1) = N_0
%p(2) = R
%p(3) = K
%
%Define data
%
pop;
%
N(1) = p(1);
error(1) = (N(1)-pops(1))^2;
for k = 1:210
N(k+1) = p(2)*N(k)/(1 + ((p(2)-1)/p(3))*N(k));
if rem(k,10)==0 %Each decade
error(1+k/10)=(N(k+1)-pops(1+k/10))^2;
end
end
totalerror = sum(error);
```

First, observe in *bherror.m* that the parameters have been stored in a parameter vector p, with three components $p(1) = N_0$, $p(2) = R$, and $p(3) = K$. The command *pop* simply calls our file *pop.m*, defining the data. The statement $N(1) = p(1)$ sets our initial population to N_0, while the line *error(1) = (N(1)-pops(1))^2* defines the error at time 0. (MATLAB does not allow an index of 0, so $k = 1$ corresponds with year 0.) The for-loop now solves the Beverton-Holt recursion iteratively, and each ten years we compute a value for our error. In order to specify that the error is only calculated once every ten years, we have the line *if rem(k,10)==0*, in which *rem(k,10)* computes the remainder of the division $k/10$, and we only compute an error when there is no remainder; that is, when k is a multiple of 10. Finally, in the line *totalerror = sum(error)*, we sum all these errors, giving E. We next minimize this function with MATLAB's function *fminsearch*. The minimization is carried out in *bhrun.m*, which also plots our model along with the data.

```
%BHRUN: MATLAB script M-file to run Beverton-Holt
%parameter estimation example.
guess = [3.93 1 1000];
```

```
[p,error]=fminsearch(@bherror, guess)
%Here, error is our usual error function E
%The rest of the file simply plots the result
pop;
years = 0:210;
N(1) = p(1);
for k = 1:210
N(k+1) = p(2)*N(k)/(1 + ((p(2)-1)/p(3))*N(k));
end
plot(decades,pops,'o',years,N)
```

With these files in place, we can now estimate our parameters and plot the result from the MATLAB command window.

```
>>bhrun
p =
7.7461 1.0217 445.9693
error =
440.9983
```

We observe that the initial population for our fit is 7.7461 million, significantly higher than the 1790 census, suggesting (reasonably) that early census counts were incomplete. The value $R = 1.0217$ indicates that the U.S. population has increased by about 2% (.0217) each year during this period. Finally, according to this model, the carrying capacity for the U. S. is roughly 446 million people. (This last value varies considerably depending on model.) The error of 440.9983 might look a bit large, but keep in mind that this keeps track of 22 data points, and the error for each is squared. That is, a more reasonable figure to look as is

$$\frac{\sqrt{440.9983}}{22} = .9545,$$

indicating that our model is off by less than 1 million people per year. The Figure associated with our fit is given as Figure 22.

10.1 Assignments

1. [10 pts] Given the U. S. population data in Table 3, find values for N_0, R, and K in the discrete logistic equation

$$N_{t+1} = N_t\Big[1 + R(1 - \frac{N_t}{K})\Big]; \quad N(0) = N_0.$$

Your unit of time should be 1 year. Turn in a plot of your model along with the data points, similar to Figure 22.

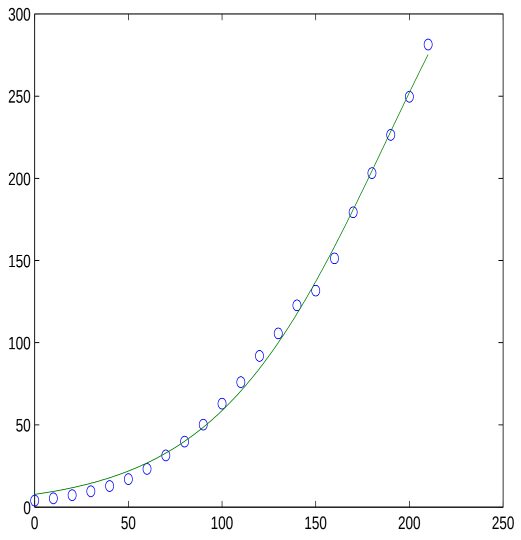


Figure 22: The Beverton-Holt model for U.S. population data.

11 Parameter Estimation Directly from ODE, Part 1[8]

In many cases we have an ODE model for a certain phenomena, but the ODE cannot be solved analytically (i.e., symbolically), even in MATLAB. In such cases, we need a method for estimating parameters from data without having a solution to the ODE.

11.1 Derivative Approximation Method

Example 11.1. Consider the *Lotka–Volterra* predator–prey model,

$$\begin{aligned}\frac{dx}{dt} &= ax - bxy\\ \frac{dy}{dt} &= -ry + cxy,\end{aligned} \tag{15}$$

where $x(t)$ denotes the population of prey at time t and $y(t)$ denotes the population of predators at time t. If we try to solve this equation symbolically in MATLAB, we find:

```
>>dsolve('Dx=a*x-b*x*y','Dy=-r*y+c*x*y');
Warning: Explicit solution could not be found; implicit solution returned.
> In <a href="error:/usr/local/Matlab2007B/toolbox/symbolic/dsolve.m,315,1">dsolve
at 315</a>
```

We will proceed, then, without ever finding a symbolic solution.

[8]The next two sections, Sections 10 and 11, assume familiarity with the method of least squares regression discussed in Sections 13 and 14 of the MATLAB notes from fall semester, *MATLAB for M151*.

Though certainly instructive to study, the Lotka–Volterra model is typically too simple to capture the complex dynamics of species interaction. One famous example that it does model fairly well is the interaction between lynx (a type of wildcat) and hare (mammals in the same biological family as rabbits), as measured by pelts collected by the Hudson Bay Company between 1900 and 1920. Raw data from the Hudson Bay Company is given in Table 6.

Year	Lynx	Hare
1900	4.0	30.0
1901	6.1	47.2
1902	9.8	70.2
1903	35.2	77.4
1904	59.4	36.3
1905	41.7	20.6
1906	19.0	18.1
1907	13.0	21.4
1908	8.3	22.0
1909	9.1	25.4
1910	7.4	27.1
1911	8.0	40.3
1912	12.3	57.0
1913	19.5	76.6
1914	45.7	52.3
1915	51.1	19.5
1916	29.7	11.2
1917	15.8	7.6
1918	9.7	14.6
1919	10.1	16.2
1920	8.6	24.7

Table 6: Number of pelts collected by the Hudson Bay Company (in 1000's).

Our goal here will be to estimate values of a, b, r, and c without finding an exact solution to (15). Beginning with the predator equation, we first assume the predator population is not zero and re-write it as

$$\frac{1}{y}\frac{dy}{dt} = cx - r.$$

If we now treat the expression $\frac{1}{y}\frac{dy}{dt}$ as a single variable, we see that c and r are respectively the slope and intercept of a line. That is, we would like to plot values of $\frac{1}{y}\frac{dy}{dt}$ versus x and fit a line through this data. Since we have a table of values for x and y, the only difficulty in this is finding values of $\frac{dy}{dt}$. In order to do this, we first recall the definition of derivative,

$$\frac{dy}{dt}(t) = \lim_{h \to 0} \frac{y(t+h) - y(t)}{h}.$$

Following the idea behind Euler's method for numerically solving differential equations, we conclude that for h sufficiently small,

$$\frac{dy}{dt}(t) \cong \frac{y(t+h) - y(t)}{h},$$

which we will call the *forward difference* derivative approximation.

Critical questions become, how good an approximation is this and can we do better? To answer the first, we recall that the Taylor polynomial of order 1 with remainder is given by

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

for some c between a and x. (A precise statement of Taylor's theorem on polynomial expansion is included in Section 5.) Letting $x = t + h$ and $a = t$, we obtain the expansion,

$$f(t+h) = f(t) + f'(t)h + \frac{f''(c)}{2}h^2,$$

where c is between t and $t+h$. Finally, we subtract $f(t)$ from both sides and divide by h to arrive at our approximation,

$$\frac{f(t+h)-f(t)}{h} = f'(t) + \frac{f''(c)}{2}h,$$

from which we see that if $f'(t)$ remains bounded for t in the domain of interest the error in our approximation will be bounded by some constant multiplied by h. We will say, then, that the forward difference derivative approximation is an *order one* approximation, and write

$$f'(t) = \frac{f(t+h)-f(t)}{h} + \mathbf{O}(|h|); \quad t \text{ typically confined to some bounded interval, } t \in [a,b],$$

where $g(h) = \mathbf{O}(|h|)$ simply means that $|\frac{g(h)}{h}|$ remains bounded as $h \to 0$, or equivalently there exists some constant C so that for $|h|$ sufficiently small $|g(h)| \leq C|h|$. More generally, we write $g(h) = \mathbf{O}(f(h))$ if $|\frac{g(h)}{f(h)}|$ remains bounded as $h \to 0$, or equivalently if there exists some constant C so that for $|h|$ sufficiently small $|g(h)| \leq C|f(h)|$.

Our second question above was, can we do better? In fact, it's not difficult to show (see homework) that the *central difference* derivative approximation is second order:

$$f'(t) = \frac{f(t+h)-f(t-h)}{2h} + \mathbf{O}(h^2).$$

Returning to our data, we observe that h in our case will be 1, not particularly small. But keep in mind that our goal is to estimate the parameters, and we can always check the validity of our estimates by checking the model against our data. (Also, as we will see in the next section, the approximations obtained by this method can be viewed as first approximations that can be improved on by the methods of Section 11.) Since we cannot compute a central difference derivative approximation for our first year's data (we don't have $y(t-h)$), we begin in 1901, and compute

$$\frac{1}{y(t)}\frac{dy}{dt} \cong \frac{1}{y(t)}\frac{y(t+h)-y(t-h)}{2h} = \frac{1}{6.1}\frac{9.8-4.0}{2} = c \cdot 47.2 - r.$$

Repeating for each year up to 1919 we obtain the system of equations that we will solve by regression. In MATLAB, the computation becomes,

```
>>lvdata
>>for k=1:19
Y(k)=(1/L(k+1))*(L(k+2)-L(k))/2;
X(k)=H(k+1);
end
>>plot(X,Y,'o')
```

From Figure 23, we can read off our first two parameter values $c = .023$ and $r = .76$. Proceeding similarly for the prey equation, we find $a = .47$ and $b = .024$, and define our model in the M-file *lv.m*. (Notice that the file *lv.m* should be edited from the form used in Section 9. In particular, the parameter values need to be changed.)

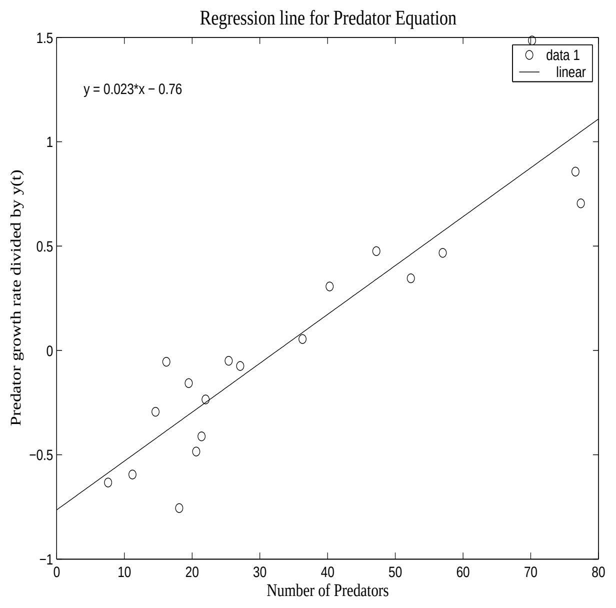


Figure 23: Linear fit for parameter estimation in prey equation.

```
function yprime = lv(t,y)
%LV: Contains Lotka-Volterra equations
a = .47; b = .024; c = .023; r = .76;
yprime = [a*y(1)-b*y(1)*y(2);-r*y(2)+c*y(1)*y(2)];
```

Finally, we check our model with Figure 24.

```
>>[t,y]=ode45(@lv,[0 20],[30 4]);
>>subplot(2,1,1);
>>plot(t,y(:,1),years,H,'o')
>>subplot(2,1,2)
>>plot(t,y(:,2),years,L,'o')
```

11.2 Assignments

1. [2 pts] Show that if $f(t)$ and its first two derivatives are all continuous on the interval $[t-h, t+h]$ and $f'''(t)$ exists and is bounded on $(t-h, t+h)$ then

$$\frac{f(t+h) - f(t-h)}{2h} = f'(t) + \mathbf{O}(h^2).$$

2. The SIR epidemic model regards the three quantities listed below:

S(t): The *susceptibles*; people who do not yet have the disease but may contract it;

I(t): The *infectives*; people who have the disease and can transmit it;

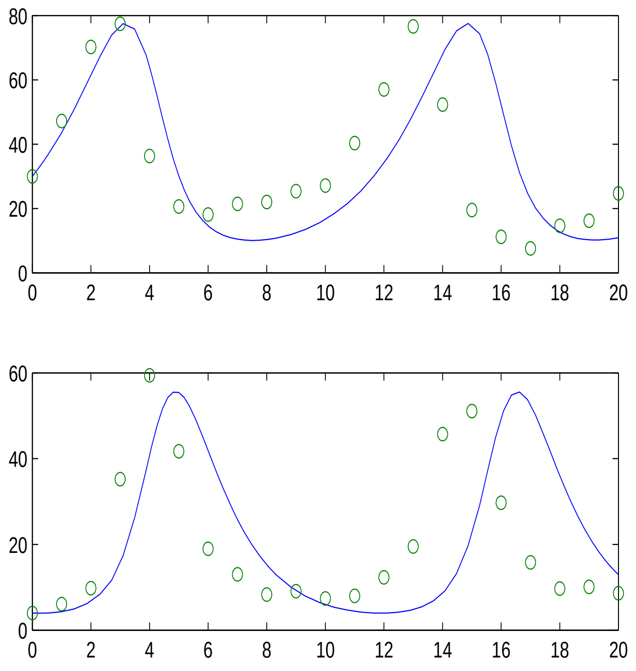


Figure 24: Model and data for Lynx–Hare example.

R(t): The *removed*; people who can neither contract nor transmit the disease. People immune to the disease fall into this category, as do people who have already died.

The SIR model describes the relationship between these quantities by the equations,

$$\begin{aligned} \frac{dS}{dt} &= -\, aSI \\ \frac{dI}{dt} &= +\, aSI - bI \\ \frac{dR}{dt} &= +\, bI. \end{aligned}$$

The March 4, 1978 issue of a journal called the *British Medical Journal* reported on an influenza epidemic at a boys boarding school. The school had 763 students, and the following data was (more or less) observed:

Day	0	3	4	5	6	7	8	9	10	11	12	13	14
S(t)	762	740	650	400	250	120	80	50	20	18	15	13	10
I(t)	1	20	80	220	300	260	240	190	120	80	20	5	2

2a. [2 pts] Use the central difference derivative approximation to determine values for the parameters a and b. (Notice that the first day you will be able to use is Day 4.)

2b. [4 pts] Create a stacked plot of your solutions $S(t)$ and $I(t)$ to the SIR model, each plotted along with the corresponding data.

2c. [2 pts] Compute the error

$$\text{Error} = \sqrt{\sum_{d=3}^{14} (S_{\text{model}}(d) - S_{\text{data}}(d))^2}.$$

12 Parameter Estimation Directly from ODE, Part 2

12.1 The Direct Method

The two primary advantages of the derivative approximation method toward parameter estimation are that (1) the computation time is typically quite reasonable, and (2) the approach can be applied with no a priori knowledge about values for the parameters. The primary drawback is that the derivative approximations we're forced to make are often quite crude. Here, we consider a second method of parameter estimation for ODE for which we carry out a regression argument directly with the ODE. In general, the method works as follows. (Warning: The next couple of lines will make a lot more sense once you've worked through Example 11.1.) Suppose, we have a system of ODE

$$\frac{d\vec{y}}{dt} = f(t, \vec{y}; \vec{p}), \quad \vec{y} \in \mathbb{R}^n, \vec{f} \in \mathbb{R}^n,$$

where $\vec{p}$ is some vector of parameters $\vec{p} \in \mathbb{R}^m$, and a collection of k data points: $(t_1, \vec{y}_1)$, $(t_2, \vec{y}_2)$,...,$(t_k, \vec{y}_k)$. In this notation, for example, the Lotka–Volterra model takes the form

$$\begin{aligned} y_1' &= p_1 y_1 - p_2 y_1 y_2 \\ y_2' &= -p_4 y_2 + p_3 y_1 y_2, \end{aligned}$$

where

$$\vec{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}; \quad \vec{p} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix}; \quad \vec{f} = \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} p_1 y_1 - p_2 y_1 y_2 \\ -p_4 y_2 + p_3 y_1 y_2 \end{pmatrix}.$$

(Notice particularly that a subscript on a vector y indicates a data point, whereas a subscript on a normal y indicates a component.)

We can now determine optimal values for the parameters $\vec{p}$ by minimizing the error,

$$E(\vec{p}) = \sum_{j=1}^{k} |\vec{y}(t_j; \vec{p}) - \vec{y}_j|^2,$$

where $|\cdot|$ denotes standard Euclidean vector norm,

$$|\vec{y}| = \sqrt{y_1^2 + \cdots + y_n^2}.$$

Example 12.1. Let's revisit the equations of Example 10.1 with this new method, so we can compare the final results. First, we write a MATLAB function M-file that contains the Lotka–Volterra model. In system form, we take $y_1(t)$ to be prey population and $y_2(t)$ to be predator population. We record the vector of parameters $\vec{p}$ as $p = (p(1), p(2), p(3), p(4)) = (a, b, c, r)$.

```
function value = lvpe(t,y,p)
%LVPE: ODE for example Lotka-Volterra parameter
%estimation example. p(1)=a, p(2) = b, p(3) = c, p(4) = r.
value=[p(1)*y(1)-p(2)*y(1)*y(2);-p(4)*y(2)+p(3)*y(1)*y(2)];
```

We now develop a MATLAB M-file that takes as input possible values of the parameter vector $\vec{p}$ and returns the squared error $E(\vec{p})$. Observe in particular here that in our *ode45()* statement we have given MATLAB a vector of times (as opposed to the general case, in which we give only an initial time and a final time). In this event, MATLAB returns values of the dependent variables only at the specified times.

```
function error = lverr(p)
%LVERR: Function defining error function for
%example with Lotka-Volterra equations.
clear y;
lvdata
[t,y] = ode45(@lvpe,years,[H(1);L(1)],[],p);
value = (y(:,1)-H').^2+(y(:,2)-L').^2;
%Primes transpose data vectors H and L
error = sum(value);
```

Finally, we minimize the function defined in *lverr.m* with MATLAB's built-in nonlinear minimizing routine *fminsearch()*. Observe that we take as our initial guess the values for a, b, c, and r we found through the derivative approximation method. Often, the best way to proceed with parameter estimation is in exactly this way, using the derivative approximation method to narrow the search and the direct method to refine it. We arrive at the refined values $a = .5486$, $b = .0283$, $c = .0264$, and $r = .8375$. The error for the derivative approximation error is $E = 15528.64$, while the error for the direct method is $E = 744.79$. See Figure 25.

```
%LVPERUN: MATLAB script M-file to run Lotka-Volterra
%parameter estimation example.
guess = [.47; .024; .023; .76];
[p,error]=fminsearch(@lverr, guess)
[t,y]=ode45(@lvpe,[0,20],[30.0; 4.0],[],p);
lvdata
subplot(2,1,1)
plot(t,y(:,1),years,H,'o')
subplot(2,1,2)
plot(t,y(:,2),years,L,'o')
```

12.2 Assignments

1. [10 pts] Use our direct method of ODE parameter estimation to obtain a revised set of parameter values for the SIR epidemic model in Problem 2 of Section 11. Answer parts (2b) and (2c) from Section 11.2, using your new parameter values.

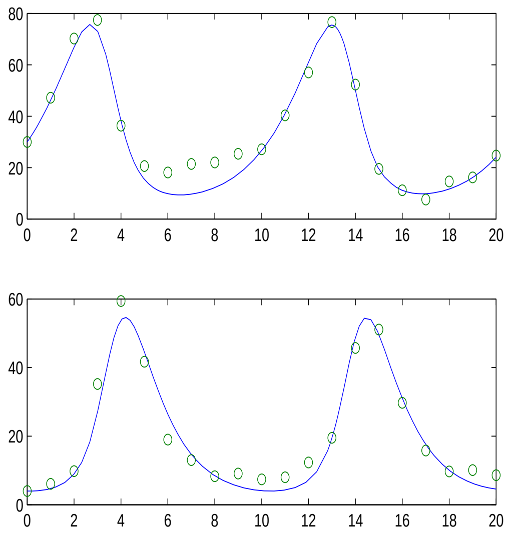


Figure 25: Refined model and data for Lynx–Hare example.

13 Direction Fields

Generally speaking, systems of nonlinear differential equations cannot be solved exactly, and so it's natural to begin thinking about what can be said about such systems in the absence of solutions. For systems of two equations a great deal of information can be obtained from a construction called the *direction field*.

13.1 Plotting Tangent Vectors

Consider the linear system of differential equations

$$\begin{aligned}\frac{dx}{dt} &= -y; \quad x(0)=1,\\ \frac{dy}{dt} &= x; \quad y(0)=0,\end{aligned} \tag{16}$$

for which it is easy to check that $(x(t), y(t)) = (\cos t, \sin t)$ is a solution. We can plot this solution pair in the x-y plane with the following MATLAB code, which produces Figure 26.

```
>>t=linspace(0,2*pi,100);
>>x=cos(t);
>>y=sin(t);
>>plot(x,y)
```

Let's consider a point on this curve, say $(x, y) = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, and determine the slope of the tangent line at this point. If we regard the top half of the curve as a function $y(x)$, then the

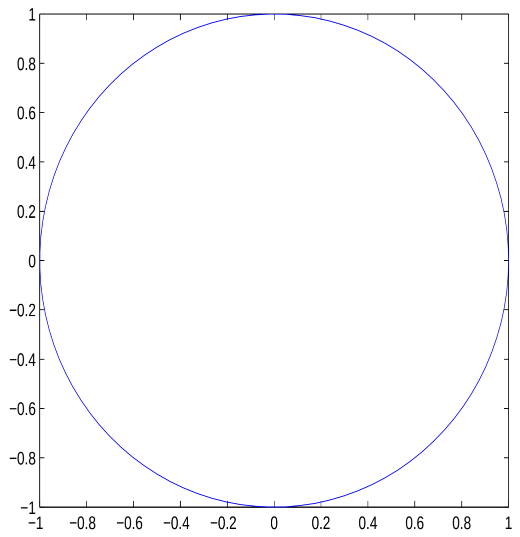


Figure 26: Plot of the curve $(\cos t, \sin t)$ for $t \in [0, 2\pi]$.

slope is $\frac{dy}{dx}$, which according to the original system of differential equations is

$$\frac{dy}{dx} = -\frac{x}{y}.$$

That is to say, the curve has the same slope as the vector $(x'(t), y'(t)) = (-y, x)$. Moreover, the direction of motion corresponds with the direction of this vector. In the case of $(x, y) = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, this tangent vector is $(x', y') = (-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. In order to see this graphically, let's add this tangent vector to our plot, setting its tail at the point $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. MATLAB's built-in M-file *quiver.m*, with usage

```
quiver(x,y,a,b)
```

places the vector (a, b) at the point (x, y). If the plot in Figure 26 is still active, this arrow can be added to it with the following MATLAB commands, which create Figure 27.

```
>>hold on
>>quiver(1/sqrt(2),1/sqrt(2),-1/sqrt(2),1/sqrt(2))
>>axis equal
```

Here, the command *axis equal* must be added while the figure is minimized.

Continuing in this manner, we can place vectors at a number of places along the curve, creating Figure 28.

```
>>quiver(-1/sqrt(2),1/sqrt(2),-1/sqrt(2),-1/sqrt(2))
>>quiver(-1/sqrt(2),-1/sqrt(2),1/sqrt(2),-1/sqrt(2))
>>quiver(1/sqrt(2),-1/sqrt(2),1/sqrt(2),1/sqrt(2))
>>quiver(0,1,-1,0)
```

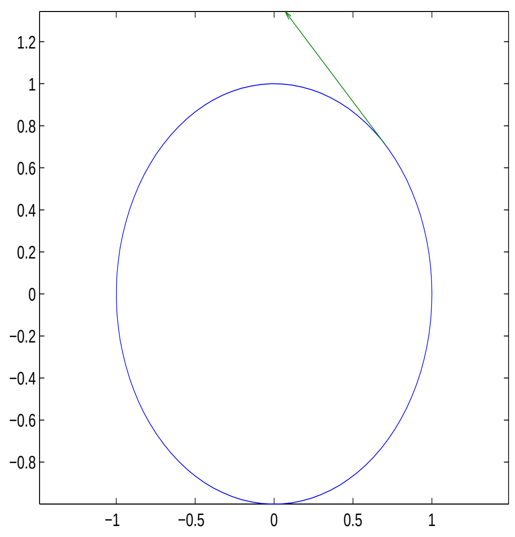


Figure 27: Plot of the curve $(\cos t, \sin t)$ for $t \in [0, 2\pi]$ along with a direction vector.

```
>>quiver(-1,0,0,-1)
>>quiver(0,-1,1,0)
>>quiver(1,0,0,1)
```

Finally, if the original circle is removed, we are left with a sketch of the direction field for this system of equations (see Figure 29). In principle this direction field consists of a collection of objects $(\vec{p}, \vec{v})$, where $\vec{p} = (x, y)$ is a point in the plane and $\vec{v} = (x', y')$ is the tangent vector emanating from that point.

Example 13.1. Plot a direction field for the Lotka–Volterra system with all coefficients taken to be 1,

$$\begin{aligned}\frac{dx}{dt} &= x - xy\\ \frac{dy}{dt} &= -y + xy.\end{aligned}$$

In this case we don't have an exact solution (see the discussion in Example 10.1), but we can still compute the direction field. We proceed by selecting certain points (x, y) in the plane and computing vectors (x', y') emanating from these points. For example, at $(1, 1)$, we have $(x', y') = (0, 0)$. At $(1, 2)$, we have $(x', y') = (-1, 0)$. Proceedingly similarly with a few more points, we can create Table 7.

We can plot these arrows with the following MATLAB code, which creates Figure 30.

```
>>axis([-1 4 -1 4])
>>hold on
>>quiver(1,2,-1,0)
>>quiver(2,1,0,1)
```

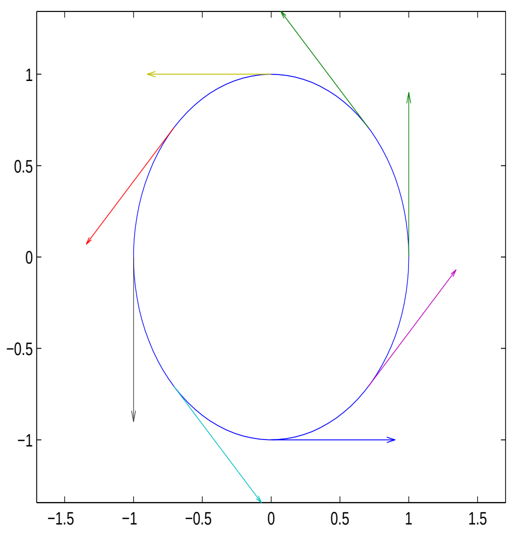


Figure 28: Plot of the curve $(\cos t, \sin t)$ for $t \in [0, 2\pi]$ along with direction vectors.

(x, y)	(x', y')
$(1, 1)$	$(0, 0)$
$(1, 2)$	$(-1, 0)$
$(2, 1)$	$(0, 1)$
$(1, 0)$	$(1, 0)$
$(0, 1)$	$(0, -1)$

Table 7: Direction arrows for the Lotka–Volterra model.

```
>>quiver(1,0,1,0)
>>quiver(0,1,0,-1)
```

The result is called the *direction field* for this equation, and though this field does not tell us the solution at each time, it tells us the direction the solution is going from any point. In other words, integral curves can be sketched in as the curves that are tangent to the vectors at each point (x, y). Clearly, the more points we place an arrow on, the more information we get. This process is easy to automate, and a nice program that does this has been written by John C. Polking, a mathematics professor at Rice University. In order to access this, open an Internet browser and go to *http://math.rice.edu/˜dfield.* Choose the file *pplane7.m* (from the M-files for MATLAB 7). From the menu at the top of your browser, choose **File**, **Save Page As**, and save this file to your home directory. Now, at the MATLAB prompt, enter *pplane7.*

In the *pplane7* setup menu, replace the default equation with the right-hand sides $x - x * y$ and $-y + x * y$ respectively, and change the minimum and maximum values of x and y so that $0 \leq x \leq 2$ and $0 \leq y \leq 2$. Select **proceed** from the bottom right corner of the setup menu, and MATLAB will draw a direction field for this equation. If you place your cursor at a point in the phase plane and select the point by left-clicking the mouse, MATLAB will draw

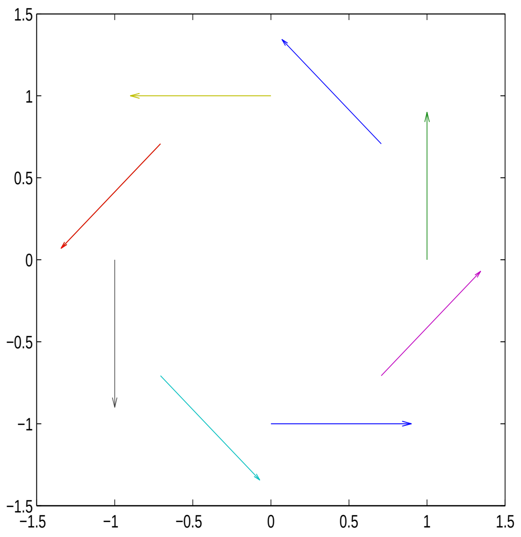


Figure 29: Direction field for the system (16).

an integral curve associated with that point. In Figure 31, three integral curves have been selected. Observe that the arrows from the direction field naturally provide the direction of movement along the integral curves.

If, instead of selecting **Arrows** from the pplane setup menu, we select **Nullclines**, we obtain Figure 32. The *nullclines* are simply the curves along which either $\frac{dx}{dt}$ or $\frac{dy}{dt}$ is 0. In this example, the nullclines are described by

$$\begin{aligned} x(1-y) &= 0 \Rightarrow x = 0 \quad \text{or} \quad y = 1 \\ y(x-1) &= 0 \Rightarrow y = 0 \quad \text{or} \quad x = 1. \end{aligned}$$

The red lines in Figure 32 correspond with the nullclines from the first equation, while the orange lines correspond with the nullclines from the second equation. (The lines are all gray in this printout, so the colors refer to the figure as it appears in MATLAB.) The arrows generically indicate the direction of motion in the regions separated by the nullclines. For example, in the region for which $x > 1$ and $y > 1$, trajectories move up and to the left (cf. Figure 31). Finally, any time a red line crosses an orange line, we must have an equilibrium point. In this case, such a crossing occurs twice: at $(0,0)$ and at $(1,1)$. $\triangle$

13.2 Assignments

1. For the Lotka–Volterra competition model

$$\begin{aligned} \frac{dx}{dt} &= r_1 x(1 - \frac{x + s_1 y}{K_1}) \\ \frac{dy}{dt} &= r_2 y(1 - \frac{s_2 x + y}{K_2}), \end{aligned}$$

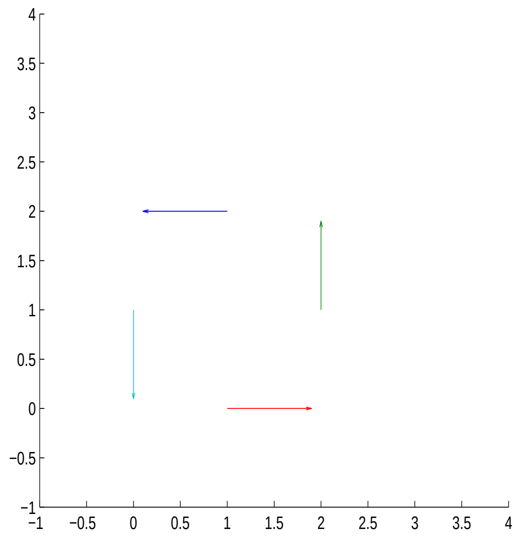


Figure 30: Direction field for the Lotka–Volterra model.

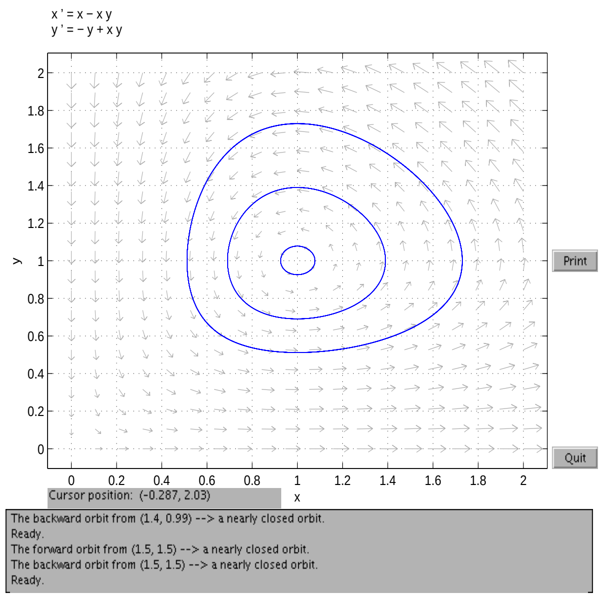
there are four cases of interest: (A) $K_1 > s_1K_2$, $K_2 < s_2K_1$, (B) $K_1 < s_1K_2$, $K_2 > s_2K_1$, (C) $K_1 > s_1K_2$, $K_2 > s_2K_1$, and (D) $K_1 < s_1K_2$, $K_2 < s_2K_1$. In this problem we will take one representative example of each of these cases. For each of the following cases, use the **Nullclines** option to find all equilibrium points for the equation (turn in the nullcline plot), and then use the **Arrows** option to determine whether or not each equilibrium point is stable or unstable (turn in the direction field plot). Give a physical interpretation of your result in each case. In all cases, take $r_1 = r_1 = 1$, and $K_1 = K_2 = 1$.

(A) [2.5 pts] $s_1 = \frac{1}{2}$, $s_2 = 2$.

(B) [2.5 pts] $s_1 = 2$, $s_2 = \frac{1}{2}$.

(C) [2.5 pts] $s_1 = \frac{1}{2}$, $s_2 = \frac{1}{4}$.

(D) [2.5 pts] $s_1 = 4$, $s_2 = 2$.



```
Cursor position:  (-0.287, 2.03)
```

```
The backward orbit from (1.4, 0.99) --> a nearly closed orbit.
Ready.
The forward orbit from (1.5, 1.5) --> a nearly closed orbit.
The backward orbit from (1.5, 1.5) --> a nearly closed orbit.
Ready.
```

Figure 31: Phase plane and direction field for the Lotka–Volterra model.

14 Eigenvalues and Stability

In this section we will use MATLAB to automate our procedure for analyzing equilibrium points for systems of ODE. The background theory in this section is fairly brutal, but the actual implementation is quite easy. Throughout the discusssion, we will be interested in autonomous systems of ODE

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}); \quad \vec{y}(0) = \vec{y}_0. \tag{17}$$

14.1 Definitions

We begin by reviewing a number of definitions that should be familiar from lecture.

Definition 14.1. (Equilibrium point) For an autonomous system of ordinary differential equations we refer to any point $\vec{y}_e$ so that $\vec{f}(\vec{y}_e) = \vec{0}$ as an *equilibrium point.*

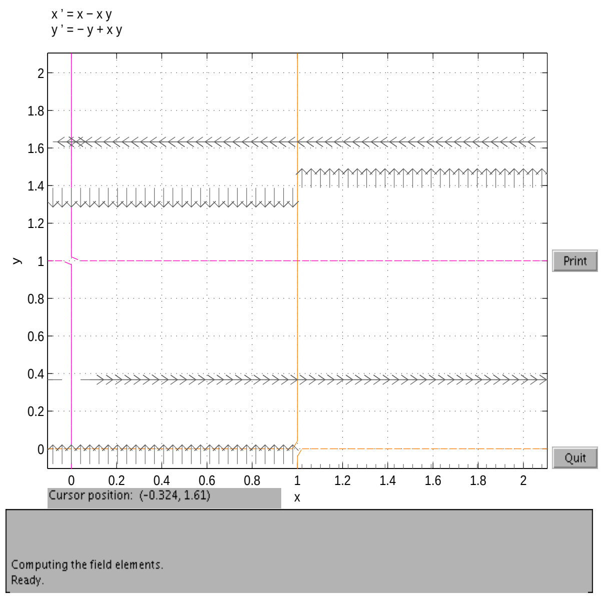
Example 14.1. Find all equilibrium points for the Lotka–Volterra system

$$\begin{aligned}\frac{dy_1}{dt} &= ay_1 - by_1y_2\\ \frac{dy_2}{dt} &= -ry_2 + cy_1y_2.\end{aligned}$$

We proceed by solving the system of algebraic equations,

$$\begin{aligned}a\hat{y}_1 - b\hat{y}_1\hat{y}_2 &= 0\\ -r\hat{y}_2 + c\hat{y}_1\hat{y}_2 &= 0,\end{aligned}$$

which has two solutions, $(\hat{y}_1, \hat{y}_2) = (0, 0)$ and $(\hat{y}_1, \hat{y}_2) = (\frac{r}{c}, \frac{a}{b})$. △



Cursor position: (-0.324, 1.61)

Computing the field elements.
Ready.

Figure 32: Nullclines for the Lotka–Volterra equation.

Definition 14.2. (Stability) An equilibrium point $\vec{y}_e$ for (17) is called *stable* if given any $\epsilon > 0$ there exists $\delta(\epsilon) > 0$ so that

$$|\vec{y}(0) - \vec{y}_e| < \delta(\epsilon) \Rightarrow |\vec{y}(t) - \vec{y}_e| < \epsilon \quad \text{for all } t > 0.$$

Definition 14.3. (Asymptotic stability.) An equilibrium point $\vec{y}_e$ for (17) is called *asymptotically stable* if it is stable and additionally there exists some $\delta_0 > 0$ so that

$$|\vec{y}(0) - \vec{y}_e| < \delta_0 \Rightarrow \lim_{t\to\infty} \vec{y}(t) = \vec{y}_e.$$

Definition 14.4. (The Jacobian matrix) For a vector function $\vec{f} \in \mathbb{R}^m$ of a vector variable $\vec{x} \in \mathbb{R}^n$, we define the $m \times n$ Jacobian matrix as

$$D\vec{f} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}. \tag{18}$$

Definition 14.5. (Differentiability.) We say that a vector function $\vec{f} \in \mathbb{R}^m$ of n variables $\vec{x} \in \mathbb{R}^n$ is differentiable at the point $\vec{x}_0$ if the first partial derivatives of all components of $\vec{f}$ all exist, and if

$$\lim_{|\vec{h}|\to 0} \frac{|\vec{f}(\vec{x}_0 + \vec{h}) - \vec{f}(\vec{x}_0) - D\vec{f}(\vec{x}_0)\vec{h}|}{|\vec{h}|} = 0.$$

Equivalently, $\vec{f}(\vec{x})$ is differentiable at $\vec{x}_0$ if there exists some $\vec{\epsilon} = \vec{\epsilon}(\vec{h})$ so that

$$\vec{f}(\vec{x}_0 + \vec{h}) = \vec{f}(\vec{x}_0) + D\vec{f}(\vec{x}_0)\vec{h} + \vec{\epsilon}(\vec{h})|\vec{h}|,$$

where $|\vec{\epsilon}| \to 0$ as $|\vec{h}| \to 0$. Here, $|\cdot|$ denotes Euclidean norm.

Notice that this definition justifies the notation

$$\vec{f}'(\vec{y}) = D\vec{f}(\vec{y}).$$

That is, the Jacobian matrix is the natural structure to define as the derivative of a such a vector function.

14.2 Linearization and the Jacobian Matrix

Suppose now that $\vec{y}_e$ denotes an equilibrium point for the system (17), and that $\vec{f}(\vec{y})$ is differentiable at $\vec{y}_e$ (see Definition 13.5). *Linearize* (17) about $\vec{y}_e$ by setting

$$\vec{y} = \vec{y}_e + \vec{z},$$

where $\vec{z}$ denotes the small *perturbation* variable (small when $\vec{y}$ is near the equilibrium point). Upon substitution of $\vec{y}$ into (17) we obtain

$$\frac{d}{dt}(\vec{y}_e + \vec{z}) = \vec{f}(\vec{y}_e + \vec{z}).$$

On the left hand side we have

$$\frac{d}{dt}(\vec{y}_e + \vec{z}) = \frac{d\vec{z}}{dt},$$

whereas on the right hand side we have (by Definition 13.5)

$$\vec{f}(\vec{y}_e + \vec{z}) = \vec{f}(\vec{y}_e) + D\vec{f}(\vec{y}_e)\vec{z} + \vec{\epsilon}(\vec{z})|\vec{z}|,$$

where $|\vec{\epsilon}| \to 0$ as $|\vec{z}| \to 0$. We observe now that $\vec{f}(\vec{y}_e) = \vec{0}$ (by definition of an equilibrium point) and $\vec{\epsilon}(\vec{z})|\vec{z}|$ is smaller than $D\vec{f}(\vec{y}_e)\vec{z}$ (since $|\vec{\epsilon}| \to 0$ as $|\vec{h}| \to 0$.) Therefore we are at least somewhat justified in taking the approximation

$$\vec{f}(\vec{y}_e + \vec{z}) \approx D\vec{f}(\vec{y}_e)\vec{z}.$$

Accordingly, we have

$$\frac{d\vec{z}}{dt} \approx D\vec{f}(\vec{y}_e)\vec{z}.$$

This is a linear system of differential equations, and we have seen in class that solutions to linear systems grow or decay depending upon the signs of the eigenvalues of the linear matrix. In applying these ideas, we will make use of the following theorem.

Theorem 14.1 (Poincare–Perron). For the ODE system

$$\frac{d\vec{y}}{dt} = \vec{f}(\vec{y}),$$

suppose $\vec{y}_e$ denotes an equilibrium point and that $\vec{f}$ is twice continuously differentiable for $\vec{y}$ in a neighborhood of $\vec{y}_e$. (I.e., all second order partial derivatives of each component of $\vec{f}$ are continuous.) Then $\vec{y}_e$ is stable or unstable as follows:

1. If the eigenvalues of $D\vec{f}(\vec{y}_e)$ all have negative real part, then $\vec{y}_e$ is asymptotically stable.

2. If any of the eigenvalues of $D\vec{f}(\vec{y}_e)$ has positive real part then $\vec{y}_e$ is unstable.

14.3 The Implementation

In practice the analysis of stability of equilibrium points consists of two steps: (1) locating the equilibrium points, and (2) finding the eigenvalues of the Jacobian matrix associated with each equilibrium point.

Example 14.2. Find all equilibrium points for the Lotka–Volterra competition model

$$\begin{aligned}\frac{dy_1}{dt} &= r_1 y_1 (1 - \frac{y_1 + s_1 y_2}{K_1})\\ \frac{dy_2}{dt} &= r_2 y_2 (1 - \frac{s_2 y_1 + y_2}{K_2}),\end{aligned}$$

and determine conditions under which each is stable.

Worked out by hand, this problem is quite long (see pp. 752–760 in the text), but here we can let MATLAB do most of the work. We use the M-file *stability.m*.

```
%STABILITY: MATLAB script M-file that locates
%equilibrium points for the Lotka-Volterra competition
%model, computes the Jacobian matrix for each
%equilibrium point, and finds the eigenvalues of
%each Jacobian matrix.
syms y1 y2 r1 K1 s1 r2 K2 s2;
y = [y1, y2];
f=[r1*y1*(1-(y1+s1*y2)/K1), r2*y2*(1-(s2*y1+y2)/K2)];
[ye1 ye2] = solve(f(1),f(2))
J = jacobian(f,y)
for k=1:length(ye1)
y1 = ye1(k)
y2 = ye2(k)
A = subs(J)
lambda = eig(A)
end
```

The only genuinely new command in this file is *jacobian*, which computes the Jacobian matrix associated with the vector function $\vec{f}$ with respect to the vector variable $\vec{y}$. The implementation is as follows.

```
>>stability
ye1 =
0
K1
0
(-K1+s1*K2)/(-1+s1*s2)
ye2 =
0
0
0
```

```
K2
-(K2-s2*K1)/(-1+s1*s2)
J =
[ r1*(1-(y1+s1*y2)/K1)-r1*y1/K1, -r1*y1*s1/K1]
[ -r2*y2*s2/K2, r2*(1-(s2*y1+y2)/K2)-r2*y2/K2]
y1 =
0
y2 =
0
A =
[ r1, 0]
[ 0, r2]
lambda =
r1
r2
y1 =
K1
y2 =
0
A =
[ -r1, -r1*s1]
[ 0, r2*(1-s2*K1/K2)]
lambda =
-r1
r2*(K2-s2*K1)/K2
y1 =
0
y2 =
K2
A =
[ r1*(1-s1*K2/K1), 0]
[ -r2*s2, -r2]
lambda =
-r1*(-K1+s1*K2)/K1
-r2
y1 =
(-K1+s1*K2)/(-1+s1*s2)
y2 =
-(K2-s2*K1)/(-1+s1*s2)
A =
[ r1*(1-((-K1+s1*K2)/(-1+s1*s2)-s1*(K2-s2*K1)/(-1+s1*s2))/K1)-r1*(-K1+s1*K2)/(-
1+s1*s2)/K1, -r1*(-K1+s1*K2)/(-1+s1*s2)*s1/K1]
[ r2*(K2-s2*K1)/(-1+s1*s2)*s2/K2, r2*(1-(s2*(-K1+s1*K2)/(-1+s1*s2)-(K2-
s2*K1)/(-1+s1*s2))/K2)+r2*(K2-s2*K1)/(-1+s1*s2)/K2]
lambda =
1/2*(-r1*s1*K2^2-K1^2*r2*s2+K1*r2*K2+r1*K1*K2
+(r1^2*s1^2*K2^4+2*r1*s1*K2^2*K1^2*r2*s2+2*r1*s1*K2^3*K1*r2-2*r1^2*s1*K2^3*K1
+K1^4*r2^2*s2^2-2*K1^3*r2^2*s2*K2+2*K1^3*r2*s2*r1*K2+K1^2*r2^2*K2^2
-2*K1^2*r2*K2^2*r1+r1^2*K1^2*K2^2+4*s1^2*K2^2*s2^2*K1^2*r1*r2
-4*s1*K2*s2^2*K1^3*r1*r2-4*s1^2*K2^3*s2*K1*r1*r2)^(1/2))/K2/K1/(-1+s1*s2)
-1/2*(r1*s1*K2^2+K1^2*r2*s2-K1*r2*K2-r1*K1*K2
+(r1^2*s1^2*K2^4+2*r1*s1*K2^2*K1^2*r2*s2+2*r1*s1*K2^3*K1*r2-2*r1^2*s1*K2^3*K1
+K1^4*r2^2*s2^2-2*K1^3*r2^2*s2*K2+2*K1^3*r2*s2*r1*K2+K1^2*r2^2*K2^2
-2*K1^2*r2*K2^2*r1+r1^2*K1^2*K2^2+4*s1^2*K2^2*s2^2*K1^2*r1*r2
-4*s1*K2*s2^2*K1^3*r1*r2-4*s1^2*K2^3*s2*K1*r1*r2)^(1/2))/K2/K1/(-1+s1*s2)
```

Reading this output from the top, we see that MATLAB has found four equilibrium points, respectively $(0,0)$, $(K_1,0)$, $(0,K_2)$, and

$$(\frac{-K_1+s_1K_2}{-1+s_1s_2}, \frac{-(K_2-s_2K_1)}{-1+s_1s_2}).$$

MATLAB associates eigenvalues with these points as follows:

$$\begin{aligned}
(0,0):&\quad \lambda = r_1, r_2\\
(K_1,0):&\quad \lambda = -r_1, \frac{r_2}{K_2}(K_2-s_2K_1)\\
(0,K_2):&\quad \lambda = -\frac{r_1}{K_1}(-K_1+s_1K_2), -r_2\\
(\frac{-K_1+s_1K_2}{-1+s_1s_2}, \frac{-(K_2-s_2K_1)}{-1+s_1s_2}):&\quad \lambda = \text{A long formula, not particularly useful.}
\end{aligned}$$

For $(0,0)$, since r_1 and r_2 are both positive, we can immediately conclude instability. For $(K_1,0)$ we have stability for $K_2 < s_2K_1$ and instability for $K_2 > s_2K_1$. For $(0,K_2)$ we have stability for $K_1 < s_1K_2$ and instability for $K_1 > s_1K_2$. For the final equilibrium point the expression MATLAB gives is too long to be useful, and we need to make a simplification. Using the general form of the Jacobian matrix (the matrix J in the MATLAB output) and the fact that this equilibrium point $(\hat{y}_1,\hat{y}_2)$ satisfies

$$\begin{aligned}
r_1 - \frac{r_1}{K_1}\hat{y}_1 - \frac{r_1s_1}{K_1}\hat{y}_2 &= 0\\
r_2 - \frac{r_2s_2}{K_2}\hat{y}_1 - \frac{r_2}{K_2}\hat{y}_2 &= 0,
\end{aligned}$$

we find that the Jacobian reduces to

$$D\vec{f}(\hat{y}_1,\hat{y}_2) = \begin{pmatrix} -\frac{r_1}{K_1}\hat{y}_1 & -\frac{r_1s_1}{K_1}\hat{y}_1 \\ -\frac{r_2s_2}{K_2}\hat{y}_2 & -\frac{r_2}{K_2}\hat{y}_2 \end{pmatrix}. \tag{19}$$

Either using MATLAB or computing by hand, we now find

$$\lambda = \frac{-(\frac{r_1\hat{y}_1}{K_1} + \frac{r_2\hat{y}_2}{K_2}) \pm \sqrt{(\frac{r_1\hat{y}_1}{K_1} + \frac{r_2\hat{y}_2}{K_2})^2 - 4\frac{r_1r_2}{K_1K_2}\hat{y}_1\hat{y}_2(1-s_1s_2)}}{2}. \tag{20}$$

We now see that the equilibrium point is stable if $(1-s_1s_2) > 0$ and unstable if $(1-s_1s_2) < 0$. △

In many cases we know specific parameter values for our model, and this serves to simplify the analysis considerably.

Example 14.3. Given the parameter values $r_1 = r_2 = K_1 = K_2 = 1$, $s_1 = \frac{1}{2}$, and $s_2 = \frac{1}{4}$, find all equilibrium points for the Lotka–Volterra competition model and determine whether or not each is stable.

Of course we could proceed simply by plugging these values into the expressions from Example 13.2, but let's instead edit our M-file so that the calculation is automated.

```
%STABILITY2: MATLAB script M-file that locates
%equilibrium points for the Lotka-Volterra competition
%model, computes the Jacobian matrix for each
%equilibrium point, and finds the eigenvalues of
%each Jacobian matrix. Parameter values are chosen
%as r1=r2=K1=K2=1, s1=.5, s2=.25.
syms y1 y2;
y = [y1, y2];
f=[y1*(1-(y1+y2/2)), y2*(1-(y1/4+y2))];
[ye1 ye2] = solve(f(1),f(2))
J = jacobian(f,y)
for k=1:length(ye1)
y1 = eval(ye1(k))
y2 = eval(ye2(k))
A = subs(J)
lambda = eig(A)
end
```

Note in particular that the parameters are no longer defined symbolically, but rather appear directly in the equations. Also, *eval* commands have been added to make y_1 and y_2 numerical. We find:

```
>>stability2
ye1 =
0
1
0
4/7
ye2 =
0
0
1
6/7
J =
[ 1-2*y1-1/2*y2, -1/2*y1]
[ -1/4*y2, 1-1/4*y1-2*y2]
y1 =
0
y2 =
0
A =
1 0
0 1
lambda =
1
1
y1 =
1
y2 =
0
A =
-1.0000 -0.5000
0 0.7500
lambda =
-1.0000
0.7500
y1 =
0
y2 =
1
A =
0.5000 0
-0.2500 -1.0000
lambda =
-1.0000
0.5000
y1 =
0.5714
y2 =
0.8571
A =
-0.5714 -0.2857
-0.2143 -0.8571
lambda =
-0.4286
-1.0000
```

In this case the equilibrium points are respectively $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(\frac{4}{7}, \frac{6}{7})$, and we can immediately conclude that the first three are unstable while the third is stable.

14.4 Assignments

1. [2 pts] Verify both (19) and (20).

2. [4 pts] Consider the system of three ODE

$$\begin{aligned}
y_1' &= ay_1(1 - \frac{y_1}{K}) - by_1y_2 \\
y_2' &= -ry_2 + cy_1y_2 - ey_2y_3 \\
y_3' &= -gy_3 + hy_2y_3,
\end{aligned}$$

which models a situation in which y_1 denotes a prey population, y_2 denotes the population of a predator that preys on y_1 and y_3 denotes the population of a predator that preys on y_2. Find all equilibrium points $(\hat{y}_1, \hat{y}_2.\hat{y}_3)$ for this system and give MATLAB expression for the eigenvalues associated with each. (You need not simplify the MATLAB expressions.)

3. [4 pts] For the system considered in Problem 2 take parameter values $a = 2$, $K = 1$, $b = 1$, $r = .5$, $c = 4$, $e = 2$, $g = 1$, and $h = 2$. Find all equilibrium points for the system with these parameters and determine whether or not each is stable.