Keith On …
Numerical Analysis

K.E. Schubert
Assistant Professor
Department of Computer Science
California State University, San Bernardino

Contents

1 Preliminaries **1**
1.1 Taylor Polynomials . . . 1
1.2 Remainder . . . 2
1.3 Multiplying Polynomials . . . 3
1.3.1 Straightforward . . . 3
1.3.2 Storing . . . 3
1.3.3 Nesting . . . 3
1.4 Binary . . . 3
1.5 Hexadecimal . . . 4
1.6 IEEE 754 . . . 6
1.7 Rounding versus Chopping . . . 7
1.8 Floating point numbers . . . 8
1.9 Propagation of Error . . . 9
1.10 Interval Arithmetic . . . 9
1.11 Condition Number . . . 10

2 Zero Finding **11**
2.1 Bisection . . . 11
2.2 Newton's Method . . . 11
2.3 Secant . . . 13
2.4 Regula Falsi . . . 16
2.5 Fixed Points . . . 16
2.6 Continuation Methods . . . 18
2.7 Multiple Roots . . . 19
2.8 Sensitivity . . . 19

3 Interpolation and Approximation **21**
3.1 Lagrange Interpolation Basis . . . 21
3.2 Divided Difference . . . 22
3.3 Error . . . 22
3.4 Splines . . . 23
3.5 Least Squares Approximation . . . 26

4 Integration **31**
4.1 Riemann . . . 31
4.2 Trapezoid . . . 33
4.3 Simpson . . . 33
4.4 Richardson . . . 33
4.5 Gaussian Quadrature . . . 35
4.6 Differentiation . . . 38

5 Differential Equations **41**
5.1 General Introduction . . . 41
5.1.1 Existence . . . 41
5.2 Euler's Method . . . 42
5.3 Runge-Kutta . . . 42
5.4 Fehlberg's Method . . . 44
5.5 Adams-Bashforth . . . 44
5.6 Adams-Moulton . . . 46
5.7 Stability & Stiff Equations . . . 47

6 Modeling and Simulation **51**
6.0.1 Electric Circuit . . . 51
6.0.2 System of Masses . . . 52
6.0.3 Linearized Pendulum . . . 52
6.0.4 Fox and Rabbits . . . 52
6.0.5 Arms race . . . 53

A Matrix Preliminaries **55**
A.1 Addition and Subtraction . . . 55
A.1.1 Using SciLab . . . 55
A.2 Multiplication . . . 57
A.2.1 Inner Product . . . 58
A.3 Scilab and Matlab Programming . . . 60
A.3.1 Matlab . . . 60

B Transform Techniques **63**
B.0.2 Laplace Transform . . . 63
B.0.3 Properties . . . 63

List of Figures

2.1 Three iterations of Newton's Method on $f(x) = x^2$ 12

3.1 Gas Thermometer Example 27
3.2 Least Squares Example 30

5.1 Existence requirements 42
5.2 Instability in Euler's Method 48
5.3 Stiff Equation 49

6.1 Foxes and Rabbits 54

List of Tables

Chapter 1

Preliminaries

1.1 Taylor Polynomials

We want an easier way of calculating a difficult function, $f(x)$. To this end we want to find a function that is similar to our original that we can calculate. Taylor polynomials, $p_n(x)$[1], are one such type of functions with an easy calculation and intuition. To find the Taylor polynomials we match the derivatives of the two polynomials at a particular point. We are in essence enforcing a smoothness criterion at the point of interest, say $x = a$.

$$\begin{aligned} p_n'(a) &= f'(a) \\ p_n''(a) &= f''(a) \\ &\vdots \\ p_n^{(n)}(a) &= f^{(n)}(a) \end{aligned}$$

Thus the general expression for the Taylor series is:

$$\begin{aligned} p_n(x) &= f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \ldots + \frac{(x-a)^n}{n!}f^{(n)}(a) \\ &= \sum_{k=0}^{n} \frac{(x-a)^k}{k!} f^{(k)}(a) \end{aligned}$$

Example

[1]The subscript n tells the highest power of the polynomial, i.e. x^n.

Problem 1.1-3(c)

$$\begin{aligned}
f(x) &= \sqrt{1+x} \\
f(0) &= \sqrt{1+0} = 1 \\
f'(0) &= \frac{1}{2\sqrt{1+0}} = \frac{1}{2} \\
f''(0) &= \frac{-1}{4(\sqrt{1+0})^3} = \frac{-1}{4} \\
f''(0) &= \frac{3}{8(\sqrt{1+0})^5} = \frac{3}{8} \\
f^{(k)}(0) &= \frac{(-1)^{k-1}(2k-3)}{2^k}
\end{aligned}$$

Example
Problem 1.1-8

$$\begin{aligned}
f(x) &= \frac{\log(1+x)}{x} \\
&\approx \frac{\sum_{k=1}^{n} \frac{(-1)^{k-1}}{k} x^k}{x} \\
&= \sum_{k=1}^{n} \frac{(-1)^{k-1}}{k} x^{k-1} \\
f(0) &\approx 1
\end{aligned}$$

1.2 Remainder

The Taylor Series obviously has errors in its approximation. If the original function is in C_{n+1} on the interval $\alpha \leqslant x \leqslant \beta$ (with a in the interval) then the remainder (or error) is given by

$$\begin{aligned}
R_n(x) &= f(x) - p_n(x) \\
&= \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(c_x)
\end{aligned}$$

with c_x between a and x. To get an error bound we assume that c_x is the worst possible.

Example
Problem 1.2-3(a) In this case $n = 1$ so the worst case would be if $\cos(c_x) = -1$ were -1.

$$R_n(x) = \frac{x^{2(1)+1}}{(2(1)+1)!} = \frac{x^3}{6} \leqslant \frac{\pi^3}{324} < 0.081$$

Example
Prove problem 8.

1.3 Multiplying Polynomials

Consider the polynomial

$$y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

1.3.1 Straightforward

The obvious way is to calculate each term separately,

$$a_k x^k = a_k * x * x * ... * x$$

This takes k multiplications for a monomial of size k, so for a polynomial with monomials up to size n it would take $\frac{n(n+1)}{2}$ multiplications.

1.3.2 Storing

Calculate $x2 = x * x$, $x3 = x * x2$, etc. This takes $2n - 1$ multiplications.

1.3.3 Nesting

Rewrite the polynomial as

$$\begin{aligned} y &= a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \\ &= ((\ldots(((a_n)x + a_{n-1})x + a_{n-2})\ldots)x + a_1)x + a_0. \end{aligned}$$

This can be done as

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= b_n x + a_{n-1} \\ b_{n-2} &= b_{n-1} x + a_{n-2} \\ &\vdots \\ b_1 &= b_2 x + a_1 \\ b_0 &= b_1 x + a_0 \end{aligned}$$

Each step takes 1 multiply so this method takes only n multiplications.

The real savings come when you have to calculate a large polynomial many times.

1.4 Binary

In any number system, the position of a digit relative to the decimal place specifies the integer power of the base we must multiple the digit by to get its value. We specify what base we are using by a subscript, if no subscript appears then the base is obvious(usually base 10, though sometimes it will be base 2 if we are calculating in base 2 for that section). So for base 10

$$101_{10} = 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0,$$

and for base 2

$$101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}.$$

This gives us one way to convert numbers. For instance, we can convert binary to decimal by expanding the binary number in this way. Thus using the above to convert binary (10.01) to decimal we find,

Note that the "2" we are using is the base of binary in decimal form, and this is why we went from binary to decimal. In binary, its form would be "10" and ten would be "1010". Therefore, we could go to binary by, expanding this out with ten in binary. The problem with this method is it is clumsy to use since we do not do squaring, cubing, etc. easily in base 2. Another problem is that 0.1 is an infinitely repeating decimal in binary so it is a pain to deal with 10-1! Instead, we convert decimal to binary as follows.

1. Split your number into a.b
2. For the whole number part, a
   (a) Divide 2 into a and note the quotient and remainder as q1,r1 (a=2*q1+r1)
   (b) As long as the quotient from above is not zero, divide it by 2 and record the quotient and remainder as qi,ri (with i denoting the current step). Repeat.
   (c) The binary equivalent of a is rnrn-1...r2r1. Basically we have done our nested polynomial evaluation backwards with x=2, and the coefficients being the remainders.
3. For the fractional part (b)
   (a) Multiply 2*b, and record the unit value as a1. Denote b-a1=b1.
   (b) If bi does not equal zero, multiply it by 2, denoting the units digit by ai+1 and the difference bi-ai+1=bi+1. Repeat until the difference is zero (this may never happen so be looking for patterns to get repeating fractions).
   (c) The fractional part, b, is a1a2a3a4...
4. The full answer is thus rnrn-1...r2r1.a1a2a3a4...

1.5 Hexadecimal

This is often made to sound more intimidating than it is. Hexadecimal numbers are simply base 16, but this can be handled nicely since 24=16. All you have to do is group binary digits into groups of 4 and use the conversion table

Bin	Hex	Dec	Bin	Hex	Dec
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

I came up with the following program in my doctoral work at UCSB.

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

int main(){
    double pi, e, result;
    int i;

    e=exp(1);

    pi=atan(1)*4;

    result=pi;

    for(i=1;i<53;i++){
        result=sqrt(result);
    }

    for(i=1;i<53;i++){
        result=result*result;
    }

    cout << setiosflags(ios::showpoint | ios::fixed) << setprecision(16);
    cout << "Pi     = " << pi << endl;
    cout << "Result = " << result << endl;
    cout << "e      = " << e << endl;

    return 0;
}
```

The results are

```
Pi     = 3.1415926535897931
Result = 2.7182818081824731
e      = 2.7182818284590451
Press any key to continue
```

Notice that Result is e to 7 significant digits, but it should be π. This underscores the importance of being numerically aware when writing programs.

1.6 IEEE 754

Floating point numbers are based off scientific notation. Consider a typical number in base 10 scientific notation,

$$-1.23 \times 10^3.$$

The number is composed of five pieces of information,

1. sign of the number (-),
2. significant or mantissa (1.23),
3. base (10),
4. sign of the exponent (+),
5. magnitude of the exponent (3).

There are two basic number formats called out in IEEE 754, single precision (float in c/c++), and double precision (double in c/c++). In addition there are two extended formats, which are only used as intermediate results while calculating.

e	f	Category	Interpretation
$1\ldots11$	$1\ldots11$ $\vdots$ $0\ldots01$	NaN	See Codes
$1\ldots11$	$0\ldots00$	$\pm\infty$	$\pm\infty$
$1\ldots10$ $\vdots$ $0\ldots01$	$1\ldots11$ $\vdots$ $0\ldots00$	Numbers	$(-1)^s \times 1.f \times 2^{(e-127)}$
$0\ldots00$	$1\ldots11$ $\vdots$ $0\ldots00$	Denormals	$(-1)^s \times 0.f \times 2^{(-126)}$
$0\ldots00$	$0\ldots00$	±0	±0

NaN codes:

Dec	Meaning	Example
1	invalid square root	$\sqrt{-1}$
2	invalid addition	$\infty + -\infty$
4	invalid division	$\frac{0}{0}$
8	invalid multiplication	$0 \times \infty$
9	invalid modulo	$x mod 0$

For this discussion, the notation $fl(x)$ will be used to mean the number x as it is represented in floating point on a computer.

$$(-1)^s \cdot 1.f \times 2^{e-127}$$

0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2
s	e								f																						

This is equivalent to saying

$$(-1)^s \cdot 1.f \times 2^{E}$$

0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	3	3	3
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2
s	e=E+127								f																						

They are the same because $e - 127 = E$ is the same equation as $e = E + 127$. I think the latter is easier to use because you read E from the number and want e. The first form (standard for most texts) involves you guessing what number produced what you are seeing (rather than calculating it). It is like trying to solve $y = mx + b$ for y given x but using the form $\frac{(y-b)}{m} = x$ to do it. It works, just not well. In any case, consider some examples.

Example:

Convert 7.892 to single precision IEEE.

Step 1: Convert 7.892 to binary

$7.892 = 111.11100100010110100000111$

Step 2: Normalize and note sign

$7.892 = (-1)^0 1.11111001000101101000000111 \times 2^2$

Step 3: Calculate Excess 127 code for exponent

$e = 2 + 127 = 129 = 10000001$

Step 4:Round $1.f$ to 24 digits

$fl(1.11111001000101101000000111) = 1.11111001000101101000100$

Step 5: Assemble

0	1 0 0 0 0 0 0 1	1 1 1 1 1 0 0 1 0 0 0 1 0 1 1 0 1 0 0 0 1 0 0

1.7 Rounding versus Chopping

Rounding is almost always used because of two reasons. To see both, let the interval between two numbers in the representation is 2δ then for rounding $x - fl(x) \in [-\delta, \delta)$, while for chopping it is $x - fl(x) \in [0, 2\delta)$. The first problem is that the error magnitude is up to twice as large for chopping. This is obviously bad, but it is not as bad as the second problem. The second problem is that all the errors of chopping have the same sign, so no error cancellation is possible when calculations are done. To see why this is bad, consider the following.

Example:

Find out the error in calculating $\sum_{i=1}^{n} x_i$ on a computer. First note that what you actually calculate is $\sum_{i=1}^{n} fl(x_i)$. The error (actual minus calculated) is thus $Err = |(\sum_{i=1}^{n} x_i) - (\sum_{i=1}^{n} fl(x_i))|$. Also let $fl(x_i) = x_i + \gamma_i$ for γ_i in the error interval of your method.

$$\begin{aligned}
Err &= \left|(\sum_{i=1}^{n} x_i) - (\sum_{i=1}^{n}(x_i + \gamma_i))\right| \\
&= \left|(\sum_{i=1}^{n} x_i) - (\sum_{i=1}^{n} x_i + \sum_{i=1}^{n} \gamma_i)\right| \\
&= \left|\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i - \sum_{i=1}^{n} \gamma_i\right| \\
&= \left|\sum_{i=1}^{n} \gamma_i\right| \\
&\leqslant \sum_{i=1}^{n} |\gamma_i|
\end{aligned}$$

For chopping the last inequality is actually an equality, i.e. chopping always has the worst case error. For a typical case on rounding the errors are distributed with some positive and some negative, thus cancellation can occur. For large sums (many terms) the law of large numbers and an assumed uniform distribution of γ_i indicates that the error for rounding will go to 0! This is a great result.

1.8 Floating point numbers

While the book discusses single precision numbers, they are essentially never used, as double precision is so much better and readily available. We will assume IEEE double precision floating point representation, as it is the standard. IEEE floating point numbers have the form , where

Single Precision Double Precision P 24 53 Emin -126 -1022 Emax 127 1023 Bias 127 1023 Thus IEEE is represented in memory as a sign bit, exponent bits (8 or 11), and mantissa bits (23 or 52). The mantissa is composed of all the bj. A few things to note about IEEE arithmetic. 1. The exponent stored is E=e-Bias 2. 0 is encoded by Emin-1 and f=0 3. Denormalized numbers are encoded by Emin-1 and f0 4. $\pm\infty$ is encoded by Emax+1 and f=0 5. NAN is encoded by Emax+1 and f0 Approximating the Reals To approximate the real number x, we define the function fl(x) as, 0 when x=0, and the nearest element in floating point to x otherwise. Finding nearest elements requires a rounding scheme (rounding or "chopping"/truncating) and a tie breaker procedure (usually round away from zero).

Bounding Errors To bound the error in approximating the real number x, we need to consider the floating point number, fl(x), used to approximate x. First we note that a real number x, is written in binary as , where s is the sign, f has as many digits as needed, and e is any integer. Note that e will be different for IEEE, which normalizes to 1fr¡2 with an implicit 1 at the start; than the non-standard forms, which normalize to 0.5 fr¡1 with no assumed leading 1. We will assume that e is within the permitted bounds for simplicity. The floating-point representation is . We can now write the difference as

For the moment, we will consider the difference (fr-f). Note that we are dealing with normalized numbers with n bits of accuracy and an implicit leading 1 (IEEE arithmetic), while the book deals with numbers normalized between a half and one, with no implicit 1, so for us . Note that the digit to the left of the decimal in f is assumed to be 1, the only exception is when fr=1.1111... which would have f=10.000...0. Technically it would actually have f=1.000...0 and the exponent would be (e+1) but since we are keeping the exponent e we keep the simplification. Note that this is equivalent to rounding 9.5 to 10. Anyway, our real concern is the worst case of the difference, which is in all cases given by . Note that the 1 is in the (n+1)st place after the decimal. We rewrite this using floating point notation as . We now stick this back into the expression for the difference between x and fl(x) and obtain an upper bound by taking absolute value . Similarly to get a lower bound we take the negative of the absolute value, and find

Now we note that the size of x is . For the book's form of the mantissa, we would have . The relative error is thus .

1.9 Propagation of Error

We have seen that representing the real numbers on a computer involves errors. When we use floating point numbers in a calculation rather than the actual numbers the errors can grow. The errors caused by using floating point approximations are called propagated errors. Two ways of bounding propagation errors exist. The forward method involves explicitly calculating the errors and is called interval arithmetic. The backward method involves finding a condition number, which gives a bound on how big the error can grow.

1.10 Interval Arithmetic

Let's consider the error in a computation between the true values (xT, yT) and the approximate values (xA, yA). We only know the approximate values and the error bounds

Note that the error is could be positive or negative so we must consider the positive and negative bounds. First, we will look at the error for addition or subtraction.

Now let's consider multiplication.

It is easy to see that this can quickly become very hard to deal with. Consider for instance multiplying two n-by-n matrices, which would involve n^3 multiplies. Keeping track of all of them would rapidly become impossible. We will consider one final operation, namely division.

Again we can see that things can become very complicated quickly.

1.11 Condition Number

We will now consider the problem of evaluating a function, f(x), at an approximate rather than true value. To do this we will require our function to be continuous on [xT,xA] and differentiable on (xT,xA). We can thus use the mean value theorem to see

We now note that since c is between the true and approximate values, and that the interval is on the order of 10-16 for IEEE double-precision arithmetic. We can thus assume c is approximately xA.

The derivative of f(x) at xA, is called the condition number and shows how the error of the approximation will influence the error of the calculation. The condition number is nice in that it cleanly handles the error bounds. It is not as precise as the error in the interval arithmetic, but it is tractable even for large matrix operations, which will involve the norms of the matrices rather than the elements. Quite a savings! Sums We have spoken a lot about summation, but we want to look at one final area of sums before we move on. Consider the following summation:

In real numbers it doesn't matter if we add the 45's first or the 100000. In floating point numbers it does matter! Floating point numbers are not associative. To see this consider a 4 decimal place accuracy machine that uses rounding, and is nicely implemented. In this case we see that 100000+45=100000 so if we add as stated we find the sum is 100000 for the series (rather than 100180). If we add the 45's first we find that 45+45+45+45=180. Then 100000+180=100200. A much better result. These sums occur in a variety of places, from standard series, to evaluating integrals, to inner products of vector, and matrix multiplication. In short you should be aware of the lack of the associative property.

Chapter 2

Zero Finding

Almost every interesting problem in mathematics can be reduced to trying to find the zeros of a function. The next several classes will be spent examining how we find zeros. In general, you cannot explicitly solve for the zeros so you need to make iterative procedures to find them. Today we will look at two methods: bisection and Newton's method.

2.1 Bisection

Bisection is a nice method in that it is guaranteed to converge and you can state exactly how many iterations it will take.

2.2 Newton's Method

Newton's Method essentially is an algebraic re-writing of the tangent line of a function at a point. Let the function we are trying to find the zero of be denoted by $f(x)$ and the point be x_i. We know that the slope of the tangent line has to be the same as the slope of the graph at the point x_i, thus the tangent line is $y = f'(x_i)x + b$. We can find b by noting the line passes through $(x_i, f(x_i))$, so

$$\begin{aligned} f(x_i) &= f'(x_i)x_i + b \\ b &= f(x_i) - f'(x_i)x_i. \end{aligned}$$

The tangent line is thus given by $y = f'(x_i)(x - x_i) + f(x_i)$. We are trying to find the zeros of the function $f(x)$ and the tangent line approximates the function so we want to find the point where the tangent line intercepts the x-axis.

$$\begin{aligned} 0 &= f'(x_i)(x_{i+1} - x_i) + f(x_i) \\ 0 &= x_{i+1} - x_i + \frac{f(x_i)}{f'(x_i)} \\ x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \end{aligned}$$

Newton's methods is thus the next estimate of the root is $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. We can see this graphically in Figure 2.1. Newton's method for $f(x) = x^2$ is thus $x_{i+1} = x_i - \frac{x_i^2}{2x_i}$ or $x_{i+1} = \frac{x_i}{2}$ The first estimate is $x_0 = 2$, the next is $x_1 = 1$, then $x_2 = .5$ and finally $x_3 = .25$. Notice that the limit is zero as desired, and we can get as close as we want by repeating the method until $x_{i+1} - x_i < tol$, where tol is some tolerance we select.

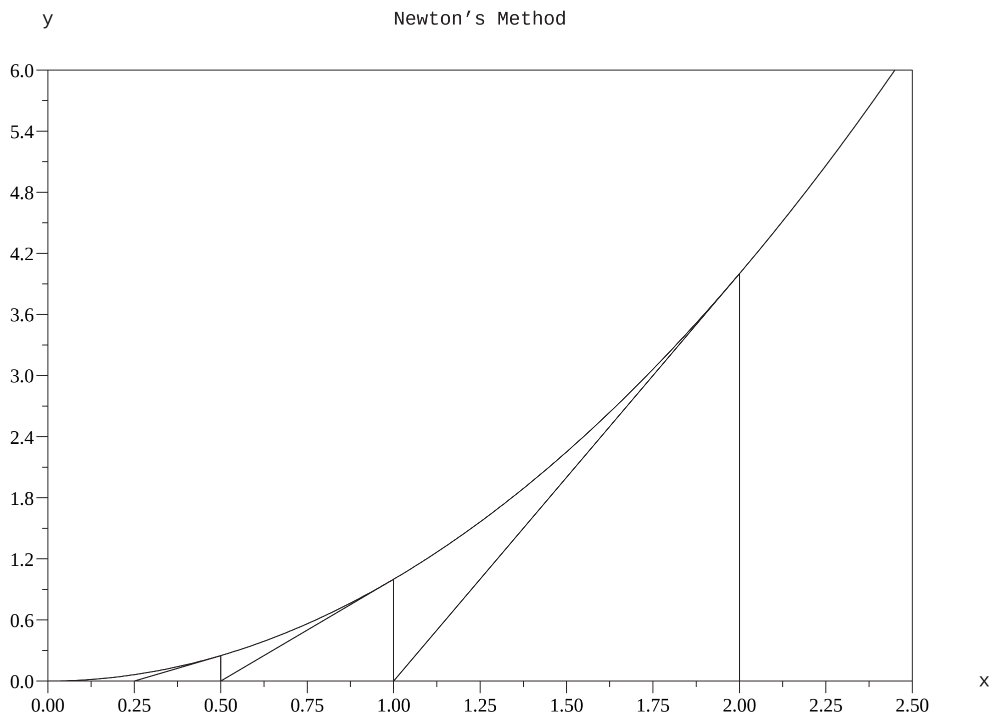

Figure 2.1: Three iterations of Newton's Method on $f(x) = x^2$

We can then use Taylor's formula to obtain an error bound. Let α be the actual value of the root.

$$\begin{aligned} e_{n+1} &= \alpha - x_{n+1} \\ &= \alpha - \left(x_n - \frac{f(x_n)}{f'(x_n)} \right) \\ &= e_n + \frac{f(\alpha) - e_n f'(\alpha) + \frac{e_n^2}{2} f''(\zeta)}{f'(x_n)} \end{aligned}$$

First, ζ is a point between α and x_n. Second, since α is a root, $f(\alpha) = 0$.

$$\begin{aligned} e_{n+1} &= e_n + \frac{f(\alpha) - e_n f'(\alpha) + \frac{e_n^2}{2} f''(\zeta)}{f'(x_n)} \\ &= e_n + \frac{-e_n f'(\alpha) + \frac{e_n^2}{2} f''(\zeta)}{f'(x_n)} \\ &= e_n \left(1 - \frac{f'(\alpha)}{f'(x_n)} \right) + e_n^2 \frac{f''(\zeta)}{2f'(x_n)} \\ &= e_n \frac{f'(x_n) - f'(\alpha)}{f'(x_n)} + e_n^2 \frac{f''(\zeta)}{2f'(x_n)} \end{aligned}$$

When x_n is close to α then $f'(x_n) - f'(\alpha) \approx 0$, so

$$\begin{aligned} e_{n+1} &= e_n^2 \frac{f''(\zeta)}{2f'(x_n)} \\ &= e_n^2 C. \end{aligned}$$

When we are close to the root, the error is dropping off as a square, thus Newton's method has quadratic convergence.

2.3 Secant

Newton's Method requires the knowledge of the first derivative of the function. Often the derivative is very complicated to evaluate and will take a long (relatively anyway) time to do so. In many cases the first derivative may not be available. In some cases it might not even exist at all points in the interval of interest. Even when it is available it could be near zero which would cause numerical problems in evaluating it, even if it is in the region of convergence. For all of these regions a new method was devised, which drew on the material leading up to calculus.

Recall that the tangent line was found as the limit of a series of secant lines. We can say that the derivative can thus be approximated by

$$f(x) \approx \frac{f(x_1) - f(x_2)}{x_1 - x_2}.$$

Thus if we know two points, we can approximate the function by a straight line between them and use the x-intercept as the next point to evaluate. We now need two points instead of one and a derivative. We refer to this as a two-point method because of the need of multiple points. We will need two estimates to begin our evaluation. Given two initial guesses, x_0 and x_1, the slope, m, is given by

$$m = \frac{f(x_1) - f(x_0)}{x_1 - x_0}.$$

Using this we find the next point, x_2 by using the point-slope form of a line

$$\begin{aligned}
f(x_2) - f(x_1) &= \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) \\
x_2 - x_1 &= \frac{x_1 - x_0}{f(x_1) - f(x_0)}(f(x_2) - f(x_1)) \\
x_2 &= x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)}.
\end{aligned}$$

We thus have the equation for the next estimate:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}. \tag{2.1}$$

Note that you can store the previous function evaluation and then you will not need to do two function evaluations per iteration.

Now we want to calculate the error. To do this we will subtract eq 2.1 from $\alpha = \alpha$.

$$\begin{aligned}
e_{n+1} &= \alpha - x_{n+1} \\
&= \alpha - \left(x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right) \\
&= \alpha - \frac{f(x_n)x_{n-1} - f(x_{n-1})x_n}{f(x_n) - f(x_{n-1})} \\
&= \frac{f(x_n)(\alpha - x_{n-1}) - f(x_{n-1})(\alpha - x_n)}{f(x_n) - f(x_{n-1})} \\
&= \frac{f(x_n)e_{n-1} - f(x_{n-1})e_n}{f(x_n) - f(x_{n-1})} \\
&= e_n e_{n-1}\frac{\frac{f(x_n)}{e_n} - \frac{f(x_{n-1})}{e_{n-1}}}{f(x_n) - f(x_{n-1})} \\
&= e_n e_{n-1}\frac{\frac{f(x_n)}{e_n} - \frac{f(x_{n-1})}{e_{n-1}}}{x_n - x_{n-1}}\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \\
&\approx e_n e_{n-1}\frac{\frac{f(x_n)}{e_n} - \frac{f(x_{n-1})}{e_{n-1}}}{x_n - x_{n-1}}\frac{1}{f'(\alpha)}
\end{aligned}$$

We need to evaluate $\frac{f(x_n)}{e_n}$, so we will use Taylor's Theorem for $f(x)$ evaluated at α. We find that

$$\begin{aligned}
\frac{f(x_n)}{e_n} &= \frac{f(\alpha) + (\alpha - x_n)f'(\alpha) + \frac{1}{2}(\alpha - x_n)^2 f''(\alpha) + \mathcal{O}((\alpha - x_n)^3)}{e_n} \\
&= \frac{e_n f'(\alpha) + \frac{1}{2}e_n^2 f''(\alpha) + \mathcal{O}(e_n^3)}{e_n} \\
&= f'(\alpha) + \frac{1}{2}e_n f''(\alpha) + \mathcal{O}(e_n^2)
\end{aligned}$$

Resuming our evaluation of e_{n+1} we find

$$\begin{aligned}
e_{n+1} &\approx e_n e_{n-1} \frac{f'(\alpha) + \frac{1}{2}e_n f''(\alpha) + \mathcal{O}(e_n^2) - f'(\alpha) - \frac{1}{2}e_{n-1}f''(\alpha) + \mathcal{O}(e_{n-1}^2)}{x_n - x_{n-1}} \frac{1}{f'(\alpha)} \\
&= e_n e_{n-1} \frac{\frac{1}{2}e_n f''(\alpha) - \frac{1}{2}e_{n-1}f''(\alpha) + \mathcal{O}(e_{n-1}^2)}{x_n - x_{n-1}} \frac{1}{f'(\alpha)} \\
&= e_n e_{n-1} \frac{\frac{1}{2}(e_n - e_{n-1})f''(\alpha) + \mathcal{O}(e_{n-1}^2)}{x_n - x_{n-1}} \frac{1}{f'(\alpha)} \\
&= e_n e_{n-1} \frac{\frac{1}{2}(x_n - x_{n-1})f''(\alpha) + \mathcal{O}(e_{n-1}^2)}{x_n - x_{n-1}} \frac{1}{f'(\alpha)} \\
&= e_n e_{n-1} (\frac{1}{2}f''(\alpha) + \mathcal{O}(e_{n-1}^2)) \frac{1}{f'(\alpha)} \\
&\approx e_n e_{n-1} \frac{f''(\alpha)}{2f'(\alpha)} \\
&\approx e_n e_{n-1} M.
\end{aligned}$$

This is similar to Newton's method which suggests that

$$e_{n+1} = A e_n^c,$$

which implies

$$\begin{aligned}
e_n &= A e_{n-1}^c \\
A^{-c^{-1}} e_n^{c^{-1}} &= e_{n-1}.
\end{aligned}$$

Substituting and collecting terms we find

$$\begin{aligned}
(A e_n^c) &= (e_n)(A^{-c^{-1}} e_n^{c^{-1}})M \\
A^{1+c^{-1}} M^{-1} &= e_n^{1-c+c^{-1}} \\
B &= e_n^{1-c+c^{-1}}.
\end{aligned}$$

Since the left hand side is a constant the exponent must be zero, because e_n is a variable.

This means

$$\begin{aligned} 0 &= 1 - c + c^{-1} \\ &= c^2 - c - 1 \\ c &= \frac{1 \pm \sqrt{1+4}}{2}, \end{aligned}$$

or c must be the golden ratio. This implies that the secant method converges superlinearly.

2.4 Regula Falsi

2.5 Fixed Points

A fixed point is a point in the domain of a function, which maps its domain back into its domain, that satisfies $\alpha = C(\alpha)$. Since α does not change when it is mapped by the function it is fixed, hence the name. We need to look at what the idea that underlies fixed points: contractions. A contraction $y = C(x)$, is a mapping from a closed interval in X into another closed interval in Y with the property that for some $b = C(a)$ (usually a and b are both the origin but it is not required), $\|\cdot\|_x$ a norm on X, and $\|\cdot\|_y$ a norm on Y we have:

$$\|x - a\|_x > \|y - b\|_y = \|C(x) - C(a)\|_y$$

for all $x \in X$ and $y \in Y$. Usually we have X and Y are $\Re$ and $a = b$, which gives us that $|x - a| > |C(x) - a|$. Take the derivative of both sides and we see

$$1 > |C'(x)|.$$

This brings up a key point, we must have that the magnitude of the function's slope is less than 1. If you think about this it makes sense, as for slope magnitudes greater than one there will be growth and we are looking a funcions which shrink things. While this is a simple idea, it has many profound implications. The book proves nicely how the uniqueness of solution, convergence, etc.. One thing that should be highlated has to do with rate of convergence. Given a contraction defined on an interval $[a, b]$ with some point, $\alpha = C(\alpha) \in [a, b]$ called a fixed point, we can define the iteration $x_{n+1} = C(x_n)$. We then have (using the mean value theorem)

$$\begin{aligned} \alpha - x_{n+1} &= C(\alpha) - C(x_n) \\ &= C'(d)(\alpha - x_n) \\ |\alpha - x_{n+1}| &< |\alpha - x_n|. \end{aligned}$$

We have linear convergence from this. Consider the following paradox.

Let a function $g(x)$ be defined by

$$g(x) = x - \frac{f(x)}{f'(x)}$$

and let $f(x)$ have a single root in some interval $[a, b]$. From the book we know this must have a fixed point in the interval and the iteration $x_{n+1} = g(x_n)$ will converge to the fixed point. This method thus has linear convergence from what we have proven above. This iteration is Newton's Method though, so it has Quadratic convergence. What gives? The convergence of a fixed point algorithm is at least linear but it can be better if $C'(\alpha) = 0$. Notice that the derivative of $g(x)$ is given by

$$\begin{aligned} g'(x) &= 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} \\ &= \frac{f(x)f''(x)}{(f'(x))^2}. \end{aligned}$$

Note that for $x = \alpha$ we trivially have that $g'(\alpha) = 0$, which satisfies our requirement for faster convergence.

How can I get a function $g(x)$ that satisfies the requirements? Many ways exist but consider the following. For a function $f(x)$ with a zero at $x = \alpha$ in an interval $[a, b]$, that has $\beta = \max_{x \in [a,b]} |f'(x)|$, we define the iteration

$$x_{n+1} = x_n - \gamma sign(f'(x_n))f(x_n)$$

with $0 < \gamma\beta < 2$. We then see that

$$\begin{aligned} g(x) &= x - \gamma sign(f'(x))f(x) \\ g'(x) &= 1 - \gamma sign(f'(x))f'(x) \\ &= 1 - \gamma|f'(x)| \end{aligned}$$

and thus $1 > g'(x) > -1$. Note that if we choose γ such that $0 < \gamma\beta < 1$ then $1 > g'(x) > 0$ and the sequence $\{x_i\}_{i=0}^{\infty}$ converges to α from one side (no alternating). The parameter γ is refered to as the **step size**. As a final note, we can use Aitken's Δ^2 method as outlined in the book to refine the estimate x_n. Replace α with $\hat{x}_n$ and you have a refinement and acceleration method that will work on any linearly convergent algorithm. It can thus be used on general fixed point methods.

Homework 4.3: 6, 13

2.6 Continuation Methods

One of the essential problems in root finding is to find a good place to start. We have spoken about the progressively doubling intervals till we find a sign change. I mentioned this was not the fastest or best, but would work. I wanted to give you what I think is one of the best. It is refered to as a continuation method or sometimes a homotopy.

A homotopy, h, is a continuous connection between two functions, f and g, that maps one space, X, to another, Y:

$$h : [0,1] \times X \to Y$$

such that $h(0,x) = g(x)$ and $h(1,x) = f(x)$. Two simple homotopies we will use are listed below.

1.
$$h(\lambda, x) = \lambda f(x) + (1-\lambda) g(x)$$

2.
$$\begin{aligned} h(\lambda, x) &= \lambda f(x) + (1-\lambda)(f(x) - f(\alpha_0)) \\ &= f(x) - (1-\lambda) f(\alpha_0) \end{aligned}$$

The first one is the most general. Assume we want to find the roots of f, but we know the roots of g. By picking a sequence of λ values from zero to one, we will slowly make the roots move from the known positions of g to the unknown positions of f. We usually try to pick g so it has the same number of roots as the function f.

The second method is a frequently used one if I don't want to find a function g. We are in essence biasing the original fucntion so that at α_0 the homotopy has a root for $\lambda = 0$. This gives a nice starting point. The following theorem tells us when this will work.

Theorem 2.1 (Ortega and Rheinboldt) *If $f : \mathbb{R}^n \to \mathbb{R}^n$ is continuously differentiable and if $\|[f'(x)]^{-1}\| \leqslant M$ on $\mathbb{R}^n$, then for any $\alpha_0 \in \mathbb{R}^n$ there is a unique curve $\{\alpha(\lambda) : 0 \leqslant \lambda \leqslant 1\}$ in $\mathbb{R}^n$ such that $f(\alpha(\lambda)) - (1-\lambda) f(\alpha_0) = 0$, with $0 \leqslant \lambda \leqslant 1$. The function $\lambda \mapsto \alpha(\lambda)$ is a continuously differentiable solution to the initial value problem $\alpha' = -[f'(\alpha)]^{-1} f(\alpha_0)$, where $\alpha(0) = \alpha_0$.*

Essentially this tells us if f is smooth and the first derivative doesn't get too close to zero then you can use this start one of our rootfinding methods, for instance Newton's Method. Often this method is solved by using a numerical integration technique which we will cover in a few weeks. For instance if Euler's method is used then it turns out to generate Newton's Method in λ!

2.7 Multiple Roots

One thing that always caused us problems in all our methods is multiple roots. I will present a simple technique for handling this case. Let our function, f, with root of multiplicity, 2, be given by

$$f(x) = (x-\alpha)^2 f_1(x),$$

where f_1 has no root at α. Take the derivative of $f(x)$ to obtain

$$\begin{aligned} f'(x) &= (x-\alpha) f_1(x) + (x-\alpha)^2 f_1'(x) \\ &= (x-\alpha)(f_1(x) + (x-\alpha) f_1'(x)) \\ &= (x-\alpha) f_2(x), \end{aligned}$$

where $f_2(x)$ has no root at α. We now have a funcition with a single root at the same place that the original function had a double root. This can be done for higher multiplicity roots, and does not require knowing α as we are taking the derivative then finding α using one of our techniques.

2.8 Sensitivity

This is refered to as stability of the roots in the books, but it is more closely related to the sensitivity of a differential equation to perturbations in its coefficients. For instance, consider a famous problem due to Wilkinson.

Problem 1 (Wilkinson) *Find the roots of the polynomial $f(x)$ given by*

$$\begin{aligned} f(x) &= (x-1)(x-2)\cdots(x-20) \\ &= x^{20} - 210x^{19} + \cdots + 20! \end{aligned}$$

The roots are clearly one through twenty. Perturb the coefficient -210 to $-210 - 2^{-23}$. The change is in one coefficient only, and that in the 7th decimal place. The roots are now

1.000000000	6.000006944	$10.095266145 \pm 0.643500904j$
2.000000000	6.999697234	$11.793633881 \pm 1.652329728j$
3.000000000	8.007267603	$13.992358137 \pm 2.518830070j$
4.000000000	8.917250249	$16.730737466 \pm 2.812624894j$
4.999999928	20.846908101	$19.502439400 \pm 1.940330347j$

The problem is not roundoff. The roots of high-order coefficients can be extremely sensitive to changes in the coefficients. This is a problem particularly when the coefficients are experimentally determined.

Chapter 3

Interpolation and Approximation

We will now look at the problem of finding a polynomial to fit a set of points. The points could come from measurements in an experiment, or it could come from a complex function we want to approximate. In either case we will begin by considering the case where we want our polynomial to be exact at these values. An obvious question is why the emphasis on polynomials, when so many other functions exist. Indeed we do see the use of other basis (sin and cos in Fourier for example), but still polynomials hold a special place in many applications. One major reason is the Theorem of Wiestrass from Real Analysis. It basically says that polynomials can approximate any function (assuming you use the entire basis).

3.1 Lagrange Interpolation Basis

Probably the nicest way to visualize the interpolation polynomials is to consider the Lagrange interpolation basis functions. For the set of points, $\{x_0, x_1, \ldots, x_n\}$ define the following polynomial:

$$L_i(x) = \frac{\prod_{j\neq i}(x - x_j)}{\prod_{j\neq i}(x_i - x_j)}.$$

We note in particular that $L_i(x_j) = \delta_{i,j}$, which allows us to get the interpolation polynomial nicely. The interpolating polynomial is then given by

$$P_n(x) = \sum_{i=0}^{n} y_i L_i(x).$$

The importance of the Lagrange basis giving us the Kronecker delta function cannot be over-emphasized, as it is the essential idea in getting the solution.

Often the points are selected to be evenly spaced due to constraints in the basic system. While this is not the best for errors, it is often a physical necessity (for example many data samplers are constrained this way). In this case we can simplify the expression using

$$\mu = \frac{x - x_0}{x_1 - x_0}.$$

This is covered well in the book.

3.2 Divided Difference

Divided difference is a similar method to Taylor approximation but instead of matching derivatives exactly at a point, it nearly approximates the derivative to exactly match certain points. The result is the same as Lagrange's formula.

$$\begin{aligned} F[x_0, x_1] &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\ F[x_0, x_1, \cdots, x_n] &= \frac{F[x_1, \cdots, x_n] - F[x_0, \cdots, x_{n-1}]}{x_n - x_0} \end{aligned}$$

$$\begin{aligned} P_1(x) &= f(x_0) + (x - x_0)F[x_0, x_1] \\ P_{k+1} &= P_k + (x - x_0)(x - x_1)\cdots(x - x_k)F[x_0, x_1, \cdots, x_{k+1}] \end{aligned}$$

Homework:
Section 5.1: 5, 9, 13
Section 5.2: 2, 3, 7

3.3 Error

The key area to note from here is that the error is given by either of the following formulas.

$$\begin{aligned} f(x) - P_n(x) &= \prod_{i=0}^{n}(x - x_i)\frac{f^{(n+1)}(c_x)}{(n+1)!} \\ &= \prod_{i=0}^{n}(x - x_i)F[x_0, x_1, \cdots, x_n, x] \end{aligned}$$

The important part of this is to note that these are themselves polynomials of order $n + 1$. Consider the plot of a polynomial with equi-spaced roots. It is trivial to note that the height of the peaks between the roots is bigger towards the outside of the interval.

3.4 Splines

For splines we want to fit a cubic polynomial for each interval so that the first and second derivatives between two sections match on the boundary. Following the books derivation we get the formula for the polynomial on the interval $[x_{j-1}, x_j]$ to be

$$\begin{aligned}
s(x) &= a_1(x_j - x)^3 + a_0(x - x_{j-1})^3 + b_1(x_j - x) + b_0(x - x_{j-1}) \\
a_i &= \frac{M_{j-i}}{6(x_j - x_{j-1})} \\
b_i &= \frac{y_{j-i} - \frac{1}{6}M_{j-i}(x_j - x_{j-1})^2}{(x_j - x_{j-1})}
\end{aligned}$$

The only thing that we need is to calculate M_i for the natural cubic spline, which is done by requiring $M_1 = M_n = 0$ and solving the following matrix system

$$\begin{aligned}
Ax &= b \\
A &= \begin{bmatrix} \alpha_2 & \beta_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_3 & \beta_3 & \ddots & \vdots \\ 0 & \beta_3 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha_{n-2} & \beta_{n-2} \\ 0 & \cdots & 0 & \beta_{n-2} & \alpha_{n-1} \end{bmatrix} \\
x &= \begin{bmatrix} M_2 \\ \vdots \\ M_{n-1} \end{bmatrix} \qquad b = \begin{bmatrix} \gamma_2 - \gamma_1 \\ \vdots \\ \gamma_{n-1} - \gamma_{n-2} \end{bmatrix} \\
\alpha_i &= \frac{x_{i+1} - x_{i-1}}{3} \qquad \beta_i = \frac{x_{i+1} - x_i}{6} \qquad \gamma_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}
\end{aligned}$$

We can also find the M_i for the not-a-knot cubic spline, which is often preferred by solving a similar system

$$\begin{aligned}
Ax &= b \\
A &= \begin{bmatrix}
\psi_1 & \beta_1 & 0 & 0 & \cdots & 0 & 0 \\
\beta_1 & \alpha_2 & \beta_2 & 0 & \cdots & 0 & 0 \\
0 & \beta_2 & \alpha_3 & \beta_3 & \ddots & \vdots & \vdots \\
0 & 0 & \beta_3 & \ddots & \ddots & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \alpha_{n-2} & \beta_{n-2} & 0 \\
0 & 0 & \cdots & 0 & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\
0 & 0 & \cdots & 0 & 0 & \beta_{n-1} & \phi_2
\end{bmatrix} \\
x &= \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} \qquad b = \begin{bmatrix} \gamma_1 - f'(x_1) \\ \gamma_2 - \gamma_1 \\ \vdots \\ \gamma_{n-1} - \gamma_{n-2} \\ f'(x_n) - \gamma_{n-1} \end{bmatrix} \\
\alpha_i &= \frac{x_{i+1} - x_{i-1}}{3} \qquad \beta_i = \frac{x_{i+1} - x_i}{6} \qquad \gamma_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \\
\psi_1 &= \frac{x_2 - x_1}{3} \qquad \phi_2 = \frac{x_n - x_{n-1}}{3}
\end{aligned}$$

or (if you don't know the derivative)

$$\begin{aligned}
Ax &= b \\
A &= \begin{bmatrix}
\psi_1 & \psi_2 & 0 & 0 & \cdots & 0 & 0 \\
\beta_1 & \alpha_2 & \beta_2 & 0 & \cdots & 0 & 0 \\
0 & \beta_2 & \alpha_3 & \beta_3 & \ddots & \vdots & \vdots \\
0 & 0 & \beta_3 & \ddots & \ddots & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \alpha_{n-2} & \beta_{n-2} & 0 \\
0 & 0 & \cdots & 0 & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\
0 & 0 & \cdots & 0 & 0 & \phi_2 & \phi_1
\end{bmatrix} \\
x &= \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-1} \\ M_n \end{bmatrix} \qquad b = \begin{bmatrix} \psi_3 \\ \gamma_2 - \gamma_1 \\ \vdots \\ \gamma_{n-1} - \gamma_{n-2} \\ \phi_3 \end{bmatrix} \\
\alpha_i &= \frac{x_{i+1} - x_{i-1}}{3} \qquad \beta_i = \frac{x_{i+1} - x_i}{6} \qquad \gamma_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \\
\xi_1 &= x_2 - x_1 \qquad \xi_2 = x_2 - z_1 \qquad \xi_3 = z_1 - x_1 \\
\psi_1 &= \frac{\xi_2^3 - \xi_1^2 \xi_2}{6\xi_1} \qquad \psi_2 = \frac{\xi_3^3 - \xi_1^2 \xi_3}{6\xi_1} \qquad \psi_3 = f(z_1) - \frac{\xi_2 y_1 + \xi_3 y_2}{\xi_1} \\
\xi_4 &= x_n - x_{n-1} \qquad \xi_5 = x_n - z_2 \qquad \xi_6 = z_2 - x_{n-1} \\
\phi_1 &= \frac{\xi_5^3 - \xi_4^2 \xi_5}{6\xi_4} \qquad \phi_2 = \frac{\xi_6^3 - \xi_4^2 \xi_6}{6\xi_4} \qquad \phi_3 = f(z_2) - \frac{\xi_5 y_{n-1} + \xi_6 y_n}{\xi_4}
\end{aligned}$$

Note, you can easily enter the matrix A into Matlab by using the command diag. For instance, if you put the entries of A that are on the main diagonal into the vector $A1$, the first sub-diagonal into $A2$, and the first super-diagonal into $A3$, then in Matlab you enter, *A=diag(A1)+diag(A2,-1)+diag(A3,1);*.

Homework

section 5.3: 7 section 5.4: 3, 5

3.5 Least Squares Approximation

Up till know we have dealt with interpolation, where we want to exactly match a set of points. In reality, we are often more concerned with having a good overall approximation rather than an exact matching at a few points. There are a lot of ways to approximate a function. In general there are two main areas discrete and continuous. We will cover the discrete case. The continuous method involves some functional analysis and we do not have the time to cover it well. If you are interested it can provide a fun project, and I have some good resources you can use.

We proceed with the discrete case. The discrete case involves measuring the function to be approximated at a series of points, and then finding the best coefficients in some sense for some functions of interest.

Some sense? What do I mean by that? Well, put simply, there are a variety of different methods of measuring how good an approximation is. The standard method is the one we will concentrate on, and it is called least squares. As with many things in Math, least squares owes its basis to Gauss. The basic idea is to reduce the sum of the squares of the distances from the measurements to the function to be fitted at each of the x values. The last point is very important because it is the basis of much of the problems in least squares. In essence the answer you get is dependent on your choice of independent variables. Below is an excerpt from my dissertation which covers what we are talking about now. The key idea to get is that there are reasons to look beyond least squares.

Consider the problem of calibrating a gas thermometer. Gas thermometers are based on Charles' law, which states that the volume of a fixed mass of gas at a fixed pressure is proportional to its temperature. A simple gas thermometer can be made by trapping some gas with a mercury plug in a capillary tube that is open on only one end [**?**]. The volume is thus proportional to the height of the plug. The equation of the thermometer is thus $hc_1 = T$, where h is the height of the plug, c_1 is the constant we want to know, and T is the absolute temperature. We place the gas thermometer in a stirred liquid bath with a known thermometer. We heat the bath and take height and temperature measurements at various times. The LS solution gives us that $\hat{c}_1 = h^\dagger T$, but we can see that this minimizes the error in the measured temperature, T, from the predicted temperature, $hh^\dagger T$. By the same token we could use the relation $h = c_2 T$, with $c_2 = \frac{1}{c_1}$. The LS solution, $\hat{c}_2 = T^\dagger h$, thus minimizes the error between the measured height, h, and the predicted height $TT^\dagger h$. A problem arises in the LS method in that generally $\hat{c}_1 \neq \frac{1}{\hat{c}_2}$. This can be seen easily in Figure 3.1. The slope of the line designated temperature errors, is $\hat{c}_1$, while the slope of the line designated height errors is $\frac{1}{\hat{c}_2}$. The line designated theoretical is the "true" system from which the estimates were generated. It is easy to see that the slopes are not the same, and thus $\hat{c}_1 \neq \frac{1}{\hat{c}_2}$. The LS solution does not even perfectly handle the case where the system matrix is "known", which gives us cause to be concerned as to how it will perform when there are perturbations to the system matrix.

The most well known alternative to least squares is total least squares (TLS). In TLS we look at the perpendicular distance to the function. This handles many of the problems of least squares but is more sensitive to errors, as it is "optimistic" in how it looks at the

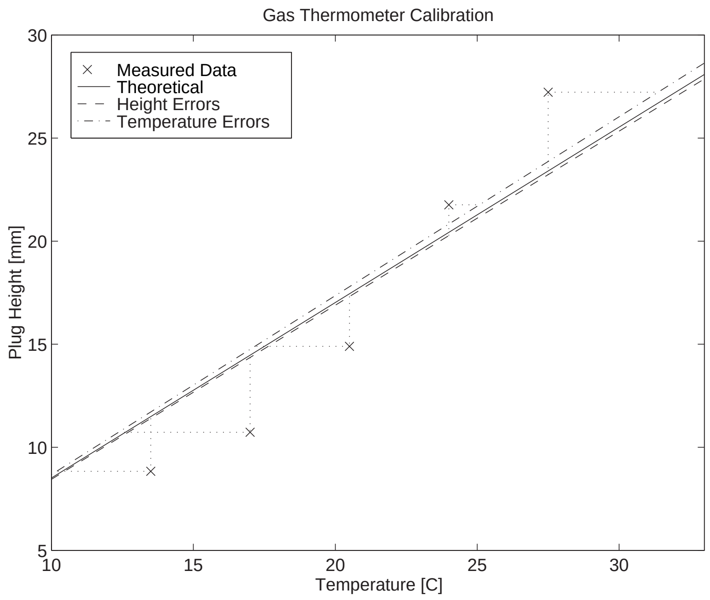

Figure 3.1: Gas Thermometer Example

problem. A huge body of literature is dedicated to this problem, and this is the central area of my dissertation. While some of these other methods are very interesting, we will stick to least squares for the moment, but we will remember that problems can occur and so if we have problems we know there are things we can do.

Getting back to business we have a set of m points (x_i, y_i) and a group of n functions $\phi_i(x)$ that we want to use to approximate the points with. We thus have m equations to find n coefficients.

$$\begin{aligned} y_1 &- \sum_{i=1}^{n} a_i \phi_i(x_1) \\ y_2 &- \sum_{i=1}^{n} a_i \phi_i(x_2) \\ &\vdots \\ y_m &- \sum_{i=1}^{n} a_i \phi_i(x_m) \end{aligned}$$

We can rewrite these into a matrix formulation, as

$$Y - \Phi A$$

where

$$\begin{aligned}
Y &= \begin{bmatrix} y_1 & y_2 & \cdots & y_m \end{bmatrix}^T \\
\Phi &= \begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_n(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_m) & \phi_2(x_m) & \cdots & \phi_n(x_m) \end{bmatrix} \\
A &= \begin{bmatrix} a_1 & a_2 & \cdots & a_m \end{bmatrix}^T .
\end{aligned}$$

At this point we want to minimize the square error which is what the 2-norm does, so we have $\min_A \|Y - \Phi A\|_2^2$. The norm we are minimizing is called the cost function. The solution is given by $A = \Phi^\dagger Y$, where $\Phi^\dagger$ is called the pseudo-inverse of Φ. Prove it? Sure! To avoid getting into some deeper areas of linear algebra we will assume that Φ has linearly independent columns. This is not restrictive, as we usually have a lot of measurements and only a few functions we want to fit to them ($m >> n$).

We recall from calculus that the minimum occurs when the gradient (derivative) is zero. We thus take the gradient of the cost with respect to A and set it equal to zero to obtain

$$\begin{aligned}
0 &= \nabla_A \|Y - \Phi A\|_2^2 \\
&= \nabla_A (Y - \Phi A)^T (Y - \Phi A) \\
&= -\Phi^T (Y - \Phi A) \\
&= \Phi^T \Phi A - \Phi^T Y \\
\Phi^T Y &= \Phi^T \Phi A
\end{aligned}$$

The last line is what is referred to as the normal equation(s). Note that some pluralize it to reflect that the single matrix equation reflects n scalar equations. I don't care, use what you like. We note that if Φ has linearly independent columns, then $(\Phi^T\Phi)^{-1}$ exists.

$$\begin{aligned}
\Phi^T \Phi A &= \Phi^T Y \\
A &= (\Phi^T \Phi)^{-1} \Phi^T Y \\
A &= \Phi^\dagger Y
\end{aligned}$$

You might wonder how the last step works. Some might just call it a definition but in reality it is because $(\Phi^T\Phi)^{-1}\Phi^T$ satisfies the four conditions of a pseudo inverse (called the Penrose conditions).

1. $\Phi\Phi^\dagger\Phi = \Phi$
2. $\Phi^\dagger\Phi\Phi^\dagger = \Phi^\dagger$
3. $\Phi\Phi^\dagger = (\Phi\Phi^\dagger)^T$
4. $\Phi^\dagger\Phi = (\Phi^\dagger\Phi)^T$

The properties are simple and easy to check, and yes, you have to check all four. Many times a candidate matrix fails only one of them. The first two properties tell us that it correctly maps the range spaces from the fundamental theorem of linear algebra, and the second two tell us the composite maps are symmetric. The pseudo-inverse always exists and is unique. Additionally, when the true inverse exists, it is the pseudo-inverse. These are just a few of the many reasons to love the pseudo-inverse…

The result is established. The nice thing about how we have handled things here is we have not specified what the functions are (they have to be linearly independent but that is no problem) or how many of them we want to fit. You can now fit any combination of functions you like.

As an example let's look at linear least squares for the points (0,1), (1,2), and (2,3). We need to find the coefficients m, and b for the line. We construct our matrices

$$\begin{aligned} Y &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^T \\ \Phi &= \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}^T \\ A &= \begin{bmatrix} b & m \end{bmatrix}^T. \end{aligned}$$

As a second example, consider fitting e^{ax} to (0,1), (1,.5), and (2,.25). To separate the coefficient, a, from the variable, x we take the natural log of $y_i = e^{ax_i}$ to obtain $\ln(y_i) = ax$. We can proceed as before now.

As a third example we will consider the second problem where we have noise (random errors) in the measurements. These three examples are coded into Matlab by

```
Y=[1;2;3];
Ye=log([1;.5;.25]);
Yee=log([1;.5;.25]+.3*rand(3,1));
X=[0;1;2];
One=ones(3,1);
Phi=[One,X];
A=Phi\Y
norm(Y-Phi*A)
Ae=X\Ye
Aee=X\Yee
Xf=0:.05:2;
Yfe=exp(Ae.*Xf);
Yfee=exp(Aee.*Xf);
p1=[-.1,2.1];
p2=[-.1,3.1];
q=[0,0];
subplot(3,1,1)
plot(X,Y,'w*',X,Phi*A,'w-',p1,q,'w-',q,p2,'w-')
axis([p1,p2])
```

```
subplot(3,1,2)
plot(X,exp(Ye),'w*',Xf,Yfe,'w-',p1,q,'w-',q,p2,'w-')
axis([p1,p2])
subplot(3,1,3)
plot(X,exp(Yee),'w*',Xf,Yfee,'w-',p1,q,'w-',q,p2,'w-')
axis([p1,p2])
```

and we get the output below and in Fig 3.2.

```
A =
    1.0000
    1.0000
ans =
    0
Ae =
   -0.6931
Aee =
   -0.4650
```

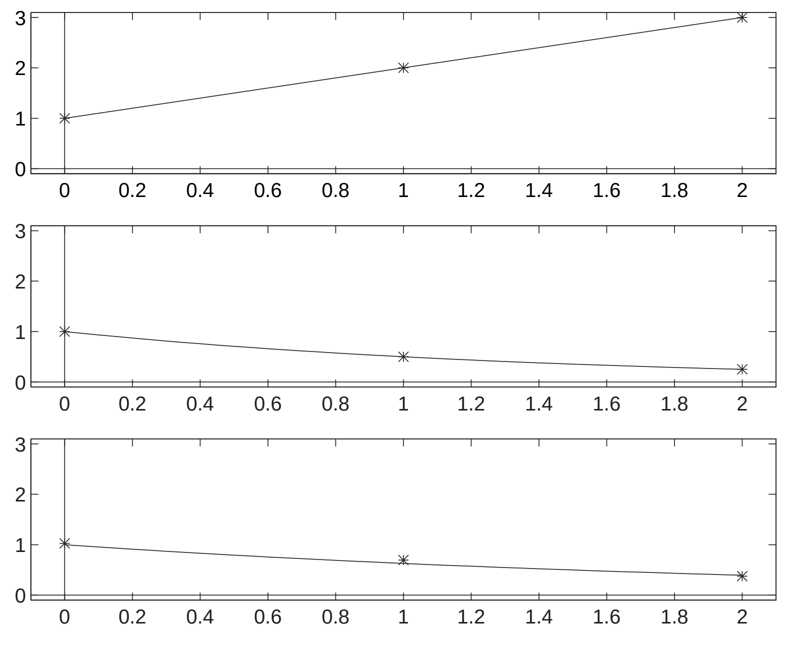

Figure 3.2: Least Squares Example

Homework 8.6: 1,3

Chapter 4

Integration

The fundamental theorem of Calculus tells us that an integral of a function can be expressed in terms of the anti-derivative of the function. Unfortunately, not all functions have anti-derivatives that are expressible in known functions. One of the most famous is the Gaussian probability distribution, which is given by

$$e^{-\left(\frac{x-\mu}{\sigma}\right)^2}.$$

The anti-derivative of this important and frequently occurring function is unknown. How do we handle it? That is the subject of this chapter.

4.1 Riemann

We recall from Calculus that the integral is defined as

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{j=1}^{n} f(p_j)(x_j - x_{j-1}).$$

Now assume that all n of the x_j are evenly spaced on $[a, b]$. We can then write

$$\begin{aligned} h &= \frac{b-a}{n} \\ &= x_j - x_{j-1}. \end{aligned}$$

We can use this to get an expression for the Riemann Sum

$$\begin{aligned}\int_a^b f(x)dx &= \lim_{n\to\infty}\sum_{j=1}^{n} f(p_j)(x_j - x_{j-1})\\ &= \lim_{n\to\infty}\sum_{j=1}^{n} f(p_j)h\\ &= \lim_{n\to\infty} h\sum_{j=1}^{n} f(p_j).\end{aligned}$$

To evaluate the integral numerically we are not able to take the limit, so we get

$$\int_a^b f(x)dx \approx h\sum_{j=1}^{n} f(p_j).$$

The exact size of n for the approximation to be good is a key aspect of numerical integration. Note also that I have not specified what p_j is, as this form allows you to do a left, right, mid-point, maximum, or minimum. The basic idea here is that we are approximating the function by a constant on the interval.

$$\int_a^b f(x)dx \approx h\sum_{j=1}^{n} f(p_j).$$

Now we want to think about the error. First we want to get an expression for the integral in terms of an exact sum. To do this we use the fundamental theorem of calculus, add nothing, rearrange, and use the mean value theorem.

$$\begin{aligned}\int_a^b f(x)dx &= F(b) - F(a)\\ &= F(x_n) - F(x_0)\\ &= F(x_n) + \sum_{i=1}^{n-1}(F(x_i) - F(x_i)) - F(x_0)\\ &= \sum_{i=1}^{n}(F(x_i) - F(x_{i-1}))\\ &= \sum_{i=1}^{n} f(c_i)(x_i - x_{i-1})\\ &= \sum_{i=1}^{n} f(c_i)h\\ &= h\sum_{i=1}^{n} f(c_i)\end{aligned}$$

This expression is exact so we can take it and subtract the expression for the midpoint method. We will assume the function has a derivative and we will note that since we have picked the midpoint that any other point in the interval is within half the width of the interval of the midpoint.

$$\begin{aligned} E &= h\sum_{i=1}^{n} f(c_i) - h\sum_{i=1}^{n} f(p_i) \\ &= h\sum_{i=1}^{n}(f(c_i) - f(p_i)) \\ &= h\sum_{i=1}^{n} f'(d_i)(c_i - p_i) \\ &\leqslant h\sum_{i=1}^{n} f'(d_i)\frac{h}{2} \\ &= \frac{h^2}{2}\sum_{i=1}^{n} f'(d_i) \end{aligned}$$

Recall that h is inversely proportional to n, so we have that the Error is inversely proportional to the square of n.

4.2 Trapezoid

$$\int_a^b f(x)dx \approx h\left(\frac{f(x_0)+f(x_n)}{2} + \sum_{j=1}^{n} f(x_j)\right)$$

4.3 Simpson

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_0)+f(x_n)+2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1})\right)$$

4.4 Richardson

Define the value of the integral to be I and the numeric approximation by some method at n node points to be I_n. We note that the error is of the form

$$\begin{aligned} E &= I - I_n \\ &= \frac{c}{n^p}, \end{aligned}$$

with c a constant dependent on the function and p a power dependent on the method. For midpoint and trapezoidal methods $p = 2$, while for Simpson's method $p = 4$. If we double the number of points then the error would be

$$\begin{aligned} E &= I - I_{2n} \\ &= \frac{c}{2^p n^p} \\ &= \frac{1}{2^p}(I - I_n). \end{aligned}$$

Solving for I we find

$$\begin{aligned} I - I_{2n} &= \frac{1}{2^p}(I - I_n) \\ I - \frac{1}{2^p} I &= I_{2n} \frac{1}{2^p} I_n \\ \frac{2^p - 1}{2^p} I &= I_{2n} - \frac{1}{2^p} I_n \\ I &= \frac{2^p}{2^p - 1} I_{2n} - \frac{1}{2^p - 1} I_n \\ I &= \frac{1}{2^p - 1}\left(2^p I_{2n} - I_n\right). \end{aligned}$$

This is Richardson's Extrapolation formula, and it will typically give a big improvement to any of the methods. We can use this estimate of the error to calculate the error

$$\begin{aligned} E &= I - I_{2n} \\ &= \frac{2^p}{2^p - 1} I_{2n} - \frac{1}{2^p - 1} I_n - I_n \\ &= \frac{2^p - (2^p - 1)}{2^p - 1} I_{2n} - \frac{1}{2^p - 1} I_n \\ &= \frac{1}{2^p - 1} I_{2n} - \frac{1}{2^p - 1} I_n \\ &= \frac{1}{2^p - 1}(I_{2n} - I_n). \end{aligned}$$

This is Richardson's error formula. We note that we can get a convergence rate by doing a little algebra on what we have already found.

$$\begin{aligned} I - I_{2n} &= \frac{1}{2^p}(I - I_n) \\ \frac{I - I_n}{I - I_{2n}} &= 2^p \end{aligned}$$

This is a nice equation but in general it is not calculable, as we don't know I and might not know p. We can handle this by considering the quantity

$$\begin{aligned}
\frac{I_{2n}-I_n}{I_{4n}-I_{2n}} &= \frac{I_{2n}-I_n+I-I}{I_{4n}-I_{2n}+I-I} \\
&= \frac{(I-I_n)-(I-I_{2n})}{(I-I_{2n})-(I-I_{4n})} \\
&= \frac{(I-I_n)-\frac{1}{2^p}(I-I_n)}{\frac{1}{2^p}(I-I_n)-\frac{1}{4^p}(I-I_n)} \\
&= \frac{1-\frac{1}{2^p}}{\frac{1}{2^p}(1-\frac{1}{2^p})} \\
&= \frac{1}{\frac{1}{2^p}} \\
&= 2^p.
\end{aligned}$$

We thus have a simple way of calculating the convergence rate, and thus a way to find p.

Homework: 7.1) 2(a), (b), (f) for midpoint, trapezoidal, and simpson. Compare with the error for trapezoidal (7.32) and Richardson's extrapolation.

7.2) 8

4.5 Gaussian Quadrature

And now for something completely different...

Last time we considered the standard way of thinking about integration, namely summing up a bunch of small areas. The technique we will discuss today was introduced in 1814 by Gauss, hence the name. We will now consider the integral

$$I(f) = \int_{-1}^{1} f(x)dx.$$

We note that this is a perfectly general statement as we note that if we define $t = (2x-a-b)/(b-a)$ then

$$\int_a^b f(x)dx = \int_{-1}^{1} f\left(\frac{(b-a)t+b+a}{2}\right)\left(\frac{b-a}{2}\right)dt$$

We have that the integral is general, if not all that obvious as to why we chose this in the first place (it will become more apparent in a few moments). We still need to show how to estimate the integral. The basic idea is to approximate the integral by weighted evaluations of the function at a series of node points. To get a good estimate we will require that our estimate at n node points will be good for every polynomial up to order $2n-1$. How? Pick

w_i and x_i such that

$$I(f) = \sum_{i=1}^{n} w_i f(x_i)$$

holds for all $f \in \{1, x, \ldots, x^{2n-1}\}$. Let's do some examples.

Consider $n = 1$.

$$\begin{aligned}
\int_{-1}^{1} dx &= w_1 f(x_1) \\
2 &= w_1 \\
\int_{-1}^{1} x dx &= w_1 x_1 \\
0 &= 2x_1 \\
0 &= x_1
\end{aligned}$$

Also, consider $n = 2$.

$$\begin{aligned}
\int_{-1}^{1} dx &= w_1 f(x_1) + w_2 f(x_2) \\
2 &= w_1 + w_2 \\
\int_{-1}^{1} x dx &= w_1 f(x_1) + w_2 f(x_2) \\
0 &= w_1 x_1 + w_2 x_2 \\
\int_{-1}^{1} x^2 dx &= w_1 f(x_1) + w_2 f(x_2) \\
\frac{2}{3} &= w_1 x_1^3 + w_2 x_2^3 \\
\int_{-1}^{1} x^3 dx &= w_1 f(x_1) + w_2 f(x_2) \\
0 &= w_1 x_1^4 + w_2 x_2^4
\end{aligned}$$

Solve these four equations for four unknowns and we find

$$\begin{aligned}
w_1 &= 1 \\
w_2 &= 1 \\
x_1 &= -\frac{1}{\sqrt{3}} \\
x_1 &= \frac{1}{\sqrt{3}}
\end{aligned}$$

The node points turn out to be the roots of the Legendre polynomials. The Legendre polynomials are defined on $[-1, 1]$ hence our choice for the limits of integration. You can find the Legendre polynomials by the following properties.

- P_n is a polynomial of order n.
- They are orthogonal

  $$\int_{-1}^{1} P_i(x)P_j(x)dx = 0$$

  when $i \neq j$.
- The normalization is

  $$\int_{-1}^{1} P_n(x)^2 dx = \frac{2}{2n+1}$$

The first several Legendre polynomials are $\{1, x, x^2 - \frac{1}{3}, x^3 - \frac{3}{5}x, x^4 - \frac{6}{7}x^2 + \frac{3}{35}\}$.

The constants can be found by

$$w_i = \int_{-1}^{1} \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} dx$$

In general we do not need to use this as the values are well tabulated.

Homework Redo 7.1) 2(a), (b), (f) for gaussian quadrature up to $n = 8$. How does the convergence compare?

4.6 Differentiation

Numerical differentiation seems obvious. In most introductions to calculus, the deriviative is introduced as the limit of a series of secant lines. Probably the most basic method of numerically taking the derivative is based off this formula. It turns out that while it seems obvious, numerical derivatives are more difficult as we will see.

Let's start at the basics, the two point method of obtaining the derivative is based off taking the derivative of the two point interpolation polynomial.

$$\begin{aligned} f(x) &\approx \frac{x_1 - x}{h} f(x_0) + \frac{x - x_0}{h} f(x_1) \\ f'(x) &\approx \frac{-1}{h} f(x_0) + \frac{1}{h} f(x_1) \\ &= \frac{f(x+h) - f(x)}{h}. \end{aligned}$$

We can see how the basic errors are dealt with by expanding the Taylor sequence.

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

Using the Taylor series in the formula we obtain that the error in the formula is

$$\begin{aligned} E &= f'(x) - \frac{f(x+h) - f(x)}{h} \\ &= f'(x) - \frac{f(x) + hf'(x) + \frac{h^2}{2} f''(c) - f(x)}{h} \\ &= f'(x) - \frac{hf'(x) + \frac{h^2}{2} f''(c)}{h} \\ &= f'(x) - f'(x) + \frac{h}{2} f''(c) \\ &= \frac{h}{2} f''(c). \end{aligned}$$

Two other types of error occur. First, there is subtractive difference errors. We note that since $f(x) \approx f(x+h)$, when we subtract them we will lose precision. Second, there is error propagation. Consider the fact that nothing we do in the real world is ever exact, so we actually measure and calculate a nearby solution:

$$\begin{aligned} f(x) &= \tilde{f}(x) + \epsilon_1 \\ f(x+h) &= \tilde{f}(x+h) + \epsilon_2. \end{aligned}$$

Substituting this into our formula we find

$$\begin{aligned} f'(x) &\approx \frac{f(x+h)-f(x)}{h} \\ &= \frac{\tilde{f}(x+h)+\epsilon_2-\tilde{f}(x)+\epsilon_1}{h} \\ &= \frac{\tilde{f}(x+h)-\tilde{f}(x)}{h}+\frac{\epsilon_2-\epsilon_1}{h}. \end{aligned}$$

The resulting error is

$$\begin{aligned} E &= f'(x)-\frac{f(x+h)-\epsilon_2-f(x)+\epsilon_1}{h} \\ &= f'(x)-\frac{f(x)+hf'(x)+\frac{h^2}{2}f''(c)-\epsilon_2-f(x)+\epsilon_1}{h} \\ &= f'(x)-\frac{hf'(x)+\frac{h^2}{2}f''(c)-\epsilon_2+\epsilon_1}{h} \\ &= f'(x)-f'(x)+\frac{h}{2}f''(c)+\frac{\epsilon_1-\epsilon_2}{h} \\ &= \frac{h}{2}f''(c)+\frac{\epsilon}{h}. \end{aligned}$$

As long as the last term is non-zero, which it will be in general, if h gets to small them the propagation errors dominate. This is the real problem. We did not have this problem with numerical integration because we were multiplying by h rather than dividing by it. The problem is more pronounced with higher order derivatives where you will be dividing by powers of h. This problem motivates the use of integral equations rather than differential ones. Unfortuneately many integral equations are very ill-conditioned. Anyway, the key idea is that there is a competition between several error types.

three point formula

Richardson

Chapter 5

Differential Equations

Most practical problems will be described by a differential equation. We will not in general know the form of the solution, but we usually can find how they change with respect to each other. From this basis we would like to be able to find the actual solution.

5.1 General Introduction

Consider the general differential equation

$$\dot{y}(x) = f(x, y(x)). \tag{5.1}$$

Using basic calculus we see that the solution is given by

$$y(x) = \int f(x, y(x))dx + c.$$

Often this is not very useful in solving equations however. The way to get practical solutions for this problem is covered in differential equations, so I will just mention a few.

5.1.1 Existence

The problem 5.1 has a solution on an interval $x_0 \leqslant x \leqslant \min(b_x, c)$ and $y_0 \leqslant y \leqslant b_y$ if the function f is continuous on the interval for $c = \frac{\|y-y_0\|}{\max_{x,y}\left(\frac{\partial f(x,y)}{\partial y}\right)}$.

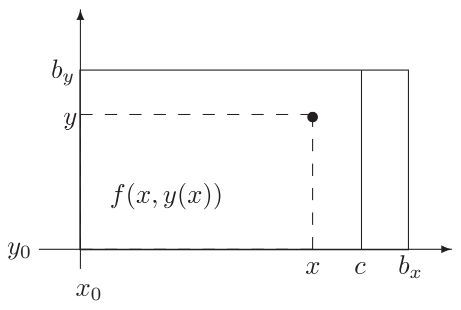

Figure 5.1: Existence requirements

5.2 Euler's Method

$$\begin{aligned} y_0 &= y(0) \\ y_1 &= y(x_0 + h) \\ &= y_0 + hf(x_0, y_0) \\ y_n &= y(x_0 + nh) \\ &= y_{n-1} + hf(x_{n-1}, y_{n-1}) \end{aligned}$$

Error is given by

$$\begin{aligned} Y(x) - y_h(x) &= hD(x) + \mathcal{O}(h^2) \\ D'(x) &= g(x)D(x) + \frac{1}{2}Y''(x), \qquad D(x_o) = 0 \\ g(x) &= \left.\frac{\partial f(x, z)}{\partial z}\right|_{z=Y(x)} \end{aligned}$$

5.3 Runge-Kutta

While Euler's method is nice and simple, it is far from the best. Higher order Taylor methods can be derived but these require evaluating multiple derivatives. Even Richardson's extrapolation has limits on its abilities.

Consider the Taylor series of y.

$$\begin{aligned}
y(x+h) &= y(x) + hy'(x) + \frac{h^2}{2}y''(x) + \dots \\
&= y(x) + hf(x,y) + \frac{h^2}{2}f'(x,y) + \dots \\
&= y(x) + hf(x,y) + \frac{h^2}{2}\left(f_x(x,y) + f_y(x,y)f(x,y)\right) + \dots \\
&= y(x) + hf(x,y) + \frac{h^2}{2}\left(f_x(x,y) + f_y(x,y)f(x,y)\right) + \dots \\
&= y(x) + hf(x,y) + ah\left(\frac{h}{2a}f_x(x,y) + \frac{h}{2a}f(x,y)f_y(x,y)\right) + \dots
\end{aligned}$$

Now we consider the Taylor series in two variables of $f(x,y)$.

$$\begin{aligned}
f(x+h, y+k) &= \sum_{n=0}^{\infty}\frac{1}{n!}\left[h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right]^n f(x,y) \\
&= f(x,y) + \left[h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right] f(x,y) + \dots \\
&= f(x,y) + hf_x(x,y) + kf_y(x,y) + \dots
\end{aligned}$$

Rearranging we find

$$f(x+h, y+k) - f(x,y) = hf_x(x,y) + kf_y(x,y) + \dots.$$

We use this in our Taylor series of y.

$$\begin{aligned}
y(x+h) &= y(x) + hf(x,y) + ah\left(\frac{h}{2a}f_x(x,y) + \frac{h}{2a}f(x,y)f_y(x,y)\right) + \dots \\
&= y(x) + hf(x,y) + h\left(af(x+\frac{h}{2a}, y + \frac{h}{2a}f(x,y)) - af(x,y)\right) + \dots \\
&= y(x) + h(1-a)f(x,y) + ahf(x+\frac{h}{2a}, y+\frac{h}{2a}f(x,y)) + \dots \\
&= y(x) + h(1-a)f(x,y) + ahf(x+b, y+bf(x,y)) + \dots \\
& \qquad b = \frac{h}{2a} \qquad 0 \leqslant a \leqslant 1
\end{aligned}$$

We have defined this in order to take advantage of the many varieties of second order R-K. The most common are: Midpoint (a=1), Modified Euler (a=1/2), and Heun's method (a=3/4).

You can also do R-K for higher orders. The most common is fourth order. The algebra is tedious so I will just present the result.

$$y(x+h) = y(x) + \frac{1}{6}\left(F_1 + 2F_2 + 2F_3 + F_4\right)$$

with

$$\begin{aligned}
F_1 &= hf(x,y)\\
F_2 &= hf(x+\frac{h}{2},y+\frac{F_1}{2})\\
F_3 &= hf(x+\frac{h}{2},y+\frac{F_2}{2})\\
F_4 &= hf(x+h,y+F_3)
\end{aligned}$$

5.4 Fehlberg's Method

We have seen that the local error (error in one step due mostly to truncation) is one order of magnitude better than the global error, in general. Often we use Richardson's Error formula to find an estimate of the local error (T_n) so we can adjust the step size to keep things nice. For instance Richardson's Error for Euler's method gives us

$$\begin{aligned}
Y(x)-y_h(x) &\approx hD(x)\\
Y(x)-y_{2h}(x) &\approx 2hD(x)\\
Y(x)-y_{2h}(x) &\approx 2(Y(x)-y_h(x))\\
Y(x) &\approx 2y_h(x)-y_{2h}(x)\\
Y(x)-y_h(x) &\approx (2y_h(x)-y_{2h}(x))-y_h(x)\\
&\approx y_h(x)-y_{2h}(x).
\end{aligned}$$

This gives us a reasonable extrapolation formula, and estimate of the error. Another way to estimate the error would be to look at two estimates from different order methods. For instance you could do a fourth and a fifth order R-K estimate at each step and use the difference to bound the error. If the error at any step became to large then you would decrease the step size and try again. This idea is Fehlberg's method, and it is the basis of most modern ode solvers. This is why Matlab has ode23 and ode45.

5.5 Adams-Bashforth

Up till now we have been looking at solving the differential equation directly. We could however just integrate both sides.

$$\begin{aligned}
y' &= f(x,y)\\
\int_{x_n}^{x_{n+1}} y'dx &= \int_{x_n}^{x_{n+1}} f(x,y)dx\\
y(x_{n+1})-y(x_n) &= \int_{x_n}^{x_{n+1}} f(x,y)dx\\
y(x_{n+1}) &= y(x_n)+\int_{x_n}^{x_{n+1}} f(x,y)dx
\end{aligned}$$

Our task is now reduced to trying to find the remaining integral, which we can use the ideas we had from last chapter. Adams-Bashforth of order m uses a polynomial approximation to $f(x,y)$ at the points x_n, x_{n-1}, $\ldots$, x_{n-m+1}. Consider the second order Adams-Bashforth method.

$$\begin{aligned}
f(x,y(x)) &\approx \frac{x_n - x}{h} f(x_{n-1}) + \frac{x - x_{n-1}}{h} f(x_n) \\
\int_{x_n}^{x_{n+1}} f(x,y)dx &\approx \int_{x_n}^{x_{n+1}} \left(\frac{x_n - x}{h} f(x_{n-1}) + \frac{x - x_{n-1}}{h} f(x_n) \right) dx \\
&\approx \frac{x_n x - \frac{1}{2}x^2}{h} f(x_{n-1}) + \frac{\frac{1}{2}x^2 - x_{n-1}x}{h} f(x_n) \Bigg|_{x_n}^{x_{n+1}} \\
&\approx \frac{x_n h - \frac{x_{n+1}^2 - x_n^2}{2}}{h} f(x_{n-1}) + \frac{\frac{x_{n+1}^2 - x_n^2}{2} - x_{n-1}h}{h} f(x_n) \\
&\approx \frac{x_n h - \frac{x_{n+1}^2 - x_{n+1}x_n + x_{n+1}x_n - x_n^2}{2}}{h} f(x_{n-1}) \\
&\qquad + \frac{\frac{x_{n+1}^2 - x_{n+1}x_n + x_{n+1}x_n - x_n^2}{2} - x_{n-1}h}{h} f(x_n) \\
&\approx \frac{x_n h - \frac{x_{n+1}h + x_n h}{2}}{h} f(x_{n-1}) + \frac{\frac{x_{n+1}h + x_n h}{2} - x_{n-1}h}{h} f(x_n) \\
&\approx \frac{2x_n - (x_{n+1} + x_n)}{2} f(x_{n-1}) + \frac{x_{n+1} + x_n - 2x_{n-1}}{2} f(x_n) \\
&\approx \frac{x_n - x_{n+1}}{2} f(x_{n-1}) + \frac{x_{n+1} - x_{n-1} + x_n - x_{n-1}}{2} f(x_n) \\
&\approx \frac{-h}{2} f(x_{n-1}) + \frac{2h + h}{2} f(x_n) \\
&\approx \frac{h}{2}(3f(x_n) - f(x_{n-1}))
\end{aligned}$$

This is kind of ugly, so it would be nice to have a faster way of handling things, especially as the dimensions increase. Luckily there is just such a technique. The method of undetermined coefficients. We start by assuming the general form we want, in this case,

$$\int_{x_n}^{x_{n+1}} f(x,y)dx \approx af(x_n) + bf(x_{n-1}$$

We would like the approximation to work perfectly for constant and linear terms so:

Constant term, $f(x,y) = 1$

$$\begin{aligned}
\int_{x_n}^{x_{n+1}} 1dx &= a \cdot 1 + b \cdot 1 \\
h &= a + b.
\end{aligned}$$

Linear term, $f(x,y) = x$

$$\begin{aligned}
\int_{x_n}^{x_{n+1}} x dx &= a \cdot x_n + b \cdot x_{n-1} \\
\frac{x_{n+1}^2 - x_n^2}{2} &= a \cdot x_n - a \cdot x_{n-1} + a \cdot x_{n-1} + b \cdot x_{n-1} \\
\frac{x_{n+1}^2 - x_n x_{n+1} + x_n x_{n+1} - x_n^2}{2} &= a \cdot (x_n - x_{n-1}) + (a+b) \cdot x_{n-1}.
\end{aligned}$$

Now noting that $a + b = h$

$$\begin{aligned}
\frac{x_{n+1}(x_{n+1} - x_n) + x_n(x_{n+1} - x_n)}{2} &= a \cdot (h) + (h) \cdot x_{n-1} \\
\frac{x_{n+1}h + x_n h}{2} &= a \cdot h + h \cdot x_{n-1} \\
\frac{x_{n+1} + x_n - 2x_{n-1}}{2} &= a \\
\frac{x_{n+1} - x_{n-1} + x_n - x_{n-1}}{2} &= a \\
\frac{2h + h}{2} &= a \\
\frac{3h}{2} &= a.
\end{aligned}$$

Thus since $a + b = h$,

$$b = \frac{-h}{2}$$

which is what we found before.

5.6 Adams-Moulton

Adams-Bashforth considered that we knew only up to $f(x,y)$ only at points up to x_n. What if we assume we can use x_n or a near approximation? Let us again consider just the simple case of linear approximations to function, though we could use any order of polynomial we liked (if we want to do the work).

$$f(x, y(x)) \approx \frac{x - x_n}{h} f(x_{n+1}) + \frac{x_{n+1} - x}{h} f(x_n)$$

Using this we can solve the integral.

$$\begin{aligned}
\int_{x_n}^{x_{n+1}} f(x,y)dx &\approx \int_{x_n}^{x_{n+1}} \left(\frac{x-x_n}{h} f(x_{n+1}) + \frac{x_{n+1}-x}{h} f(x_n)\right) dx \\
&\approx \frac{\frac{1}{2}x^2}{h} f(x_{n+1} - x_n x) + \frac{x_{n+1}x - \frac{1}{2}x^2}{h} f(x_n)\Big|_{x_n}^{x_{n+1}} \\
&\approx \frac{\frac{x_{n+1}^2 - x_n^2}{2} - x_n x_{n+1} + x_n^2}{h} f(x_{n+1}) + \frac{x_{n+1}^2 - x_{n+1}x_n - \frac{x_{n+1}^2 - x_n^2}{2}}{h} f(x_n) \\
&\approx \frac{x_{n+1}^2 - x_n^2 - 2x_n x_{n+1} + 2x_n^2}{2h} f(x_{n+1}) + \frac{2x_{n+1}^2 - 2x_{n+1}x_n - x_{n+1}^2 + x_n^2}{2h} f(x_n) \\
&\approx \frac{x_{n+1}^2 - 2x_n x_{n+1} + x_n^2}{2h} f(x_{n+1}) + \frac{x_{n+1}^2 - 2x_{n+1}x_n - +x_n^2}{2h} f(x_n) \\
&\approx \frac{(x_{n+1} - x_n)^2}{2h} f(x_{n+1}) + \frac{(x_{n+1} - x_n)^2}{2h} f(x_n) \\
&\approx \frac{h^2}{2h} f(x_{n+1}) + \frac{h^2}{2h} f(x_n) \\
&\approx \frac{h}{2}(f(x_{n+1}) + f(x_n))
\end{aligned}$$

5.7 Stability & Stiff Equations

A good start for looking at stiff equations is to examine the stability of our methods. Consider Forward Euler for

$$y' = -200y, \qquad y(1) = e^{-200}.$$

The solution can easily be seen to be e^{-200x}. Forward Euler gives us

$$\begin{aligned}
y_{i+1} &= y_i - 200 * h * y_i \\
&= y_i(1 - 200 * h).
\end{aligned}$$

We note that this is stable if $\|1 - 200 * h\| < 1$. Since $h > 0$ we must have $h < 0.01$. This is a smooth, monotonically decreasing function that is less than 2^{-87} and greater than zero. Despite the smoothness and flatness, we have to take very small steps. To see this look at Figure 5.2. Note that Backward Euler does not have the same problem. It is defined by

$$\begin{aligned}
y_{i+1} &= y_i - 200 * h * y_{i+1} \\
&= y_i \frac{1}{1 + 200 * h},
\end{aligned}$$

which is stable for all $h > 0$. Stability is not the same thing as stiffness, but they are related. Stiffness is due to multiple scales of the terms, for instance e^{-x}, e^{-200x}. Consider the following equation

$$0 = \ddot{y} + 1001\dot{y} + 1000y, \qquad y(0) = 1, \qquad \dot{y}(0) = -1.$$

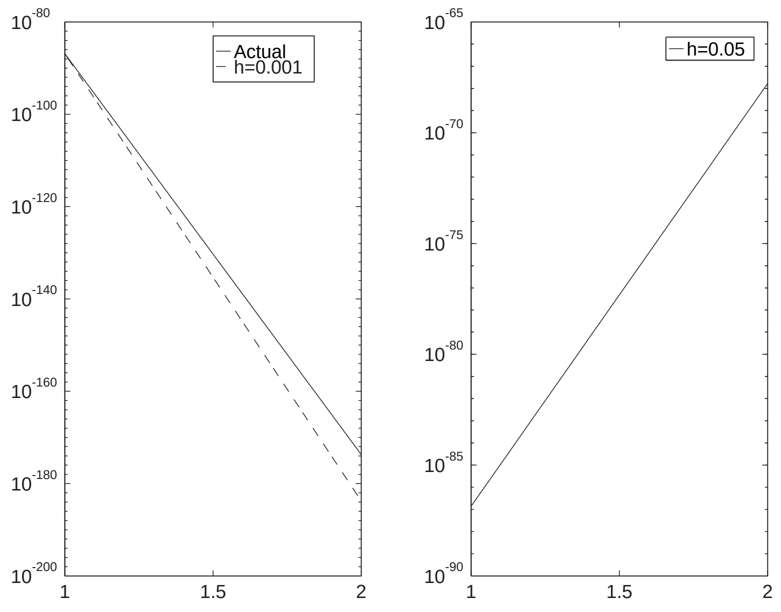

Figure 5.2: Instability in Euler's Method

The solution can easily be seen to be e^{-t}, which is nice in any definition. We solve the equation using a 4th order R-K method. The results are in Figure 5.3.

What is going on? We need to look at the eigenvalues.

Define the intermediate variable $z = \dot{y}$, and the equation becomes

$$\begin{aligned} 0 &= \dot{z} + 1001z + 1000y \\ \dot{z} &= -1001z - 1000y. \end{aligned}$$

Putting this in matrix form

$$\dot{\begin{bmatrix} y \\ z \end{bmatrix}} = \begin{bmatrix} 0 & 1 \\ -1000 & -1001 \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix}$$

The eigenvalues are (-1,-1000). Since the eigenvalues differ by three orders of magnitude we can expect stiffness, which is what Figure 5.3 shows.

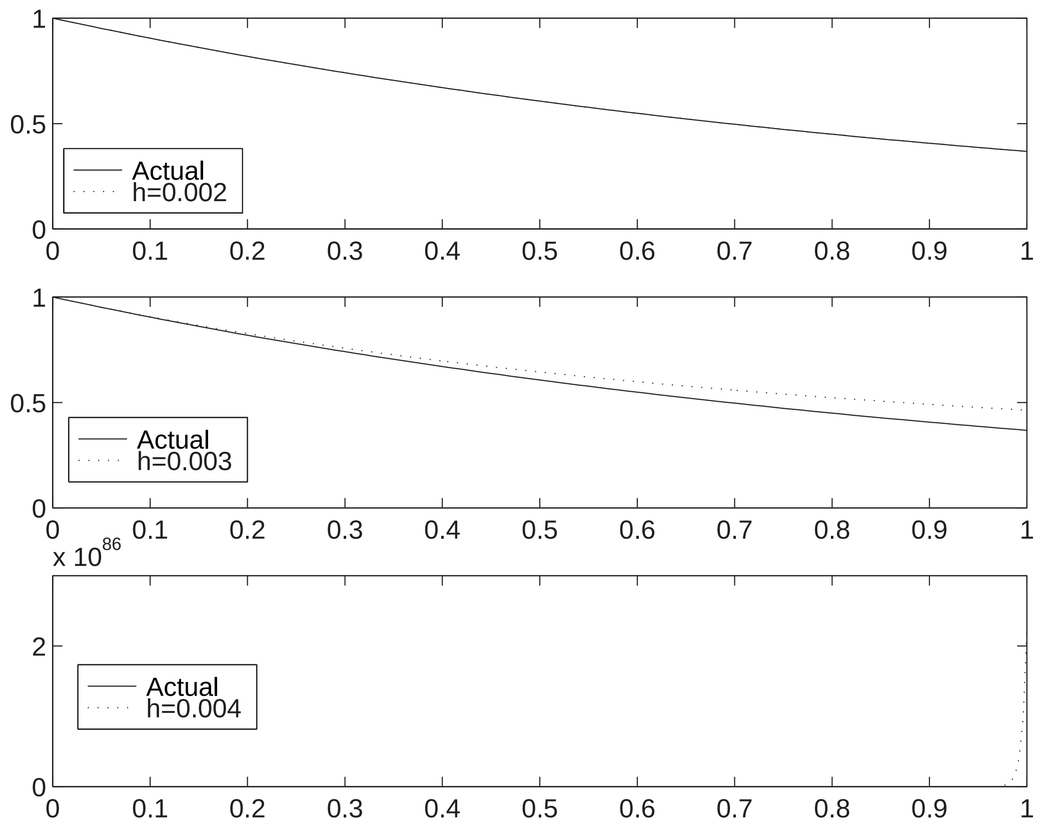

Figure 5.3: Stiff Equation

Chapter 6

Modeling and Simulation

Some systems cannot be practically tested. Consider the spread of a highly infectious disease, or the destruction due to a nuclear war. We can look at smaller examples of these, but the full blown case would be bad to run. What we would like to do is to take our mathematical knowledge and create a system of equations which will have the same behavior. We can then test the real system by seeing how the modelled system behaves. This action of making a a mathematical model be similar to the real system is simulation (same root as similar). Often these systems are expressed as either differential or difference equations. They can thus be solved by our differential equation solvers directly, or transformed (Laplace, Fourier, Z, etc.) so they can be solved by zero-finding.

Lets look at how we can set some of these up, and then give some examples.

6.0.1 Electric Circuit

Kirkoff's law tells us that the sum of the voltage drops around a loop must be zero. We can use this to figure out how to set up some models. To do so we need to note how each element operates. On the attached sheets I have included some copies of physical elements and how they relate. If we consider our basic variable as charge then current is just the time derivative of charge and we can see the elements are just:

$$\begin{aligned} V_{inductor} &= L\ddot{q} \\ V_{resistor} &= R\dot{q} \\ V_{capacitor} &= \frac{1}{C}q \end{aligned}$$

We can then write an equation for each loop and plot the first derivative of each of the loop charges (thus loop current).

6.0.2 System of Masses

Consider a system of masses connected by springs and dampers. We can note that the if we choose our variable to be position then we have:

$$\begin{aligned} F_{spring} &= -kx \\ F_{damper} &= -c\dot{x} \\ F_{mass} &= m\ddot{x} \end{aligned}$$

6.0.3 Linearized Pendulum

$$\dot{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} = \begin{bmatrix} 0 & 1 \\ -\sqrt{\frac{g}{l}} & -c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

6.0.4 Fox and Rabbits

Consider the problem where there is a predator and its prey in the area. If there are too many predators they might eat off all the prey and starve themselves. If there are too much of the prey the predators will increase in number because of the plentiful food. How do we model this?

We want several things in our model. First, predators should naturally starve, only the presence of prey offsets this. Prey should naturally breed, and predators offset this. The offset to both should be a scaled proportion of the contacts. Contact should be rare when either population is low and more frequent with a reasonable number of both. These considerations lead to the model that contacts occur at a rate proportional to the product of predators and prey. These rules are combined in the model below.

$$\begin{aligned} \dot{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} &= \begin{bmatrix} -m_1 & 0 \\ 0 & m_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 x_1 x_2 \\ -b_2 x_1 x_2 \end{bmatrix} \\ &= \begin{bmatrix} b_1 x_2 - m_1 & 0 \\ 0 & m_2 - b_2 x_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \end{aligned}$$

To keep the population constant in each we need $b_1 x_2 - m_1 = 0$ and $m_2 - b_2 x_1 = 0$, thus

$$\begin{aligned} x(1) &= \frac{m_2}{b_2} \\ x(2) &= \frac{m_1}{b_1} \end{aligned}$$

We can implement the basic predator prey rate calculations by the Scilab function

```
function [xdot]=predator_prey_rate(starve,birth,eat,eaten,predator,prey)
    xdot=zeros(2,1);
    xdot(1)=(eat*prey-starve)*predator;
    xdot(2)=(birth-eaten*predator)*prey;
endfunction
```

To specialize this to our foxes and rabbits we could write the Scilab function

```
function [xdot]=fox_rabbit(t,x)
    starve=5;
    birth=10;
    eat=.01;
    eaten=.5;
    predator=x(1);
    prey=x(2);
    getf('predator_prey_rate.sce');
    xdot=predator_prey_rate(starve,birth,eat,eaten,predator,prey);
endfunction
```

Now we need to solve it, so we will use the function ode. The following code will set everything up for us and call ode.

```
getf('fox_rabbit.sce');
x=ode([22;500],0,[0:.01:10],fox_rabbit);

//set("figure_style","new") //create a figure
subplot(211)
   plot2d([0:.01:10],x')
   xtitle('Fox and Rabbit History','Years','Population');
subplot(212)
   plot2d(x(1,:),x(2,:))
   xtitle('Fox and Rabbit Interaction','Foxes','Rabbits');
```

When we run it we get the following graphic.

6.0.5 Arms race

$$\dot{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} = \begin{bmatrix} -m_1 & a_1 \\ a_2 & -m_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 x_2^2 \\ b_2 x_1^2 \end{bmatrix}$$

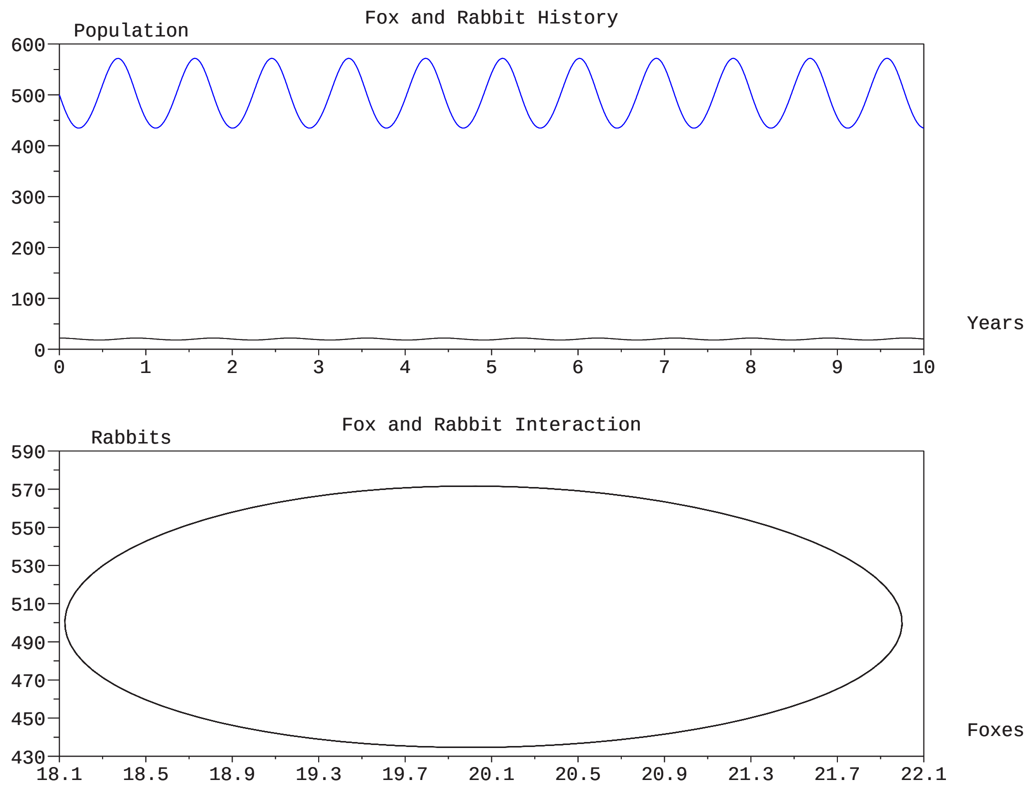

Figure 6.1: Foxes and Rabbits

Appendix A

Matrix Preliminaries

Matrices are an essential tool in control systems design. First lets discuss some basic terms.

$\mathbb{R}$ The real numbers.

$\mathbb{R}^n$ The vectors composed of n-tuples of real numbers.

$\mathbb{R}^{m\times n}$ The matrices composed of m rows and n columns of real numbers.

A.1 Addition and Subtraction

In order to add or subtract matrices, the dimensions must be the same. Essentially we can add two $\mathbb{R}^{m\times n}$ matrices but not a $\mathbb{R}^{m\times n}$ and a $\mathbb{R}^{p\times r}$ matrix or even a $\mathbb{R}^{m\times n}$ and a $\mathbb{R}^{m\times m}$ matrix. Addition (or subtraction) is done by adding (or subtracting) the corresponding elements in the two matrices. For two $\mathbb{R}^{3\times 2}$ matrices addition is given by

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1+7 & 2+8 \\ 3+9 & 4+10 \\ 5+11 & 6+12 \end{bmatrix} = \begin{bmatrix} 8 & 10 \\ 12 & 14 \\ 16 & 18 \end{bmatrix} \tag{A.1}$$

For two $\mathbb{R}^{3\times 2}$ matrices addition is given by

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} - \begin{bmatrix} 6 & 5 \\ 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1-6 & 2-5 \\ 3-4 & 4-3 \\ 5-2 & 6-1 \end{bmatrix} = \begin{bmatrix} -5 & -3 \\ -1 & 1 \\ 3 & 5 \end{bmatrix} \tag{A.2}$$

A.1.1 Using SciLab

When you first start SciLab you will see something like

```
==========
 scilab-2.7.2
```

```
        Copyright (C) 1989-2003 INRIA/ENPC
                   ==========



Startup execution:
  loading initial environment

-->
```

The arrow "$-->$" is the command prompt. SciLab, like MatLab is a command line interface to a mathematics programming environment. To get started lets do a calculation.

```
3+(2+5*4)/11
```

SciLab performs the calculation and displays the answer.

```
 ans  =

    5.
```

Now lets define a simple variable.

```
a=2
```

SciLab responds with

```
 a  =

    2.
```

Notice anything similar? The response is almost the same but "ans" has been replaced by the variable name "a". In fact it is even more similar than that. When no assignment ("name=") is given, SciLab automatically assigns the result to the variable "ans". Try using it.

```
a*ans
```

SciLab will tell you that "ans" is now 10. Lets move on and define a matrix. Type the following

```
  A=[1,2;3,4;5,6]
```

and press enter. Commas are used to separate elements and semicolons are used to separate rows. Note that you could also have entered "A" using the alternate notation

```
  A=[[1 2];[3 4];[5 6]]
```

or even (command prompt shown so you won't think something is wrong when it automatically appears, also you do not need to space over like I do to enter the numbers, I just find it easier to read)

```
-->  A=[[1 2]
-->     [3 4]
-->     [5 6]]
```

Thus spaces work like commas and returns work like semicolons. In any case, SciLab should respond by showing you that it has created the matrix variable as follows

```
 A  =

!   1.    2. !
!   3.    4. !
!   5.    6. !
```

The variable "A" is now defined and can be used. For instance we might want to define "B" to be "A+A". Do this by typing

```
  B=A+A
```

SciLab will add the matrices and define "B" to be the result, showing you the answer.

```
 B  =

!   2.     4.  !
!   6.     8.  !
!   10.    12. !
```

This mode is useful for doing simple calculations and testing output. We will refer to it as the interactive mode. Since SciLab has an interactive mode that is command driven, it is reasonable to assume it would have a programming interface (we will refer to it as the programming mode). I will show the use of programming mode later.

A.2 Multiplication

In order to do multiplication, we need to have matrices that are compatible to multiply. Recall that in order to add two matrices they had to be the same size. Unfortuneately this is not the condition for multiplication. To be able to multiply a matrix A on the right by a matrix B[1], we must have that the inner matrix dimensions are equal. That is, we require that $A \in \mathbb{R}^{m\times p}$ and $B \in \mathbb{R}^{p\times n}$, Where m, n, and p do not have to be the same, but they could. Thus we could multiply the following

[1]Multiplying a matrix A on the right by a matrix B means AB, left multiplication by B would be BA. In matrices it is not true in general that $AB = BA$, or even that one can be calculated even if the other exists. This will make more sense in a few seconds.

- AB or BA with $\{A, B\} \in \mathbb{R}^{3\times 3}$;
- AB with $A \in \mathbb{R}^{3\times 3}$ and $B \in \mathbb{R}^{3\times 4}$;
- AB with $A \in \mathbb{R}^{2\times 3}$ and $B \in \mathbb{R}^{3\times 4}$.

Two general ways to do multiplication exist. Each has computational advantages in different situations. They give the same answer they are just different ways to group the solution.

A.2.1 Inner Product

A full study of the inner product is beyond the scope of this work, but interested students are referred to texts on Hilbert Spaces[2]. An inner product of two vectors of size n[3] a and b is given by $\langle a, b\rangle = a^T b = \sum_{i=1}^{n}(a_i b_i)$. Thus it is the sum of the product of corresponding elements in the vectors.

Example:

$$a = \begin{bmatrix}1 & 2 & 3\end{bmatrix}^T \qquad b = \begin{bmatrix}4 & 5 & 6\end{bmatrix}^T$$

Note that I have defined the vectors in the transposed form to save space. The transpose just means

$$\begin{bmatrix}1 & 2\end{bmatrix}^T = \begin{bmatrix}1\\2\end{bmatrix}. \tag{A.3}$$

Then the inner product of a and b is

$$\begin{aligned}\langle a, b\rangle &= \begin{bmatrix}1 & 2 & 3\end{bmatrix}\begin{bmatrix}4\\5\\6\end{bmatrix}\\ &= 1\times 4 + 2\times 5 + 3\times 6\\ &= 32\end{aligned}$$

Now try it in SciLab. First define a and b by

```
--> a=[1;2;3];
--> b=[4;5;6];
```

Note that I ended each line with a ";", which tells SciLab to suppress its responses. This is vital when using the programming mode or SciLab will scroll out lots of unneeded info. With the vectors described we just need to take the inner product. To do this we use the $a^T b$ form, which is written

[2]Hilbert Spaces are spaces with a defined inner product. They play an important role in control systems theory

[3]A vector of size n is the same as saying the vector is in $\mathbb{R}^n$.

```
--> a'*b
```

The prime(') does the transpose (^{T}) and the multiply then does the inner product.

With vector inner products down we can consider two matrices, $A \in \mathbb{R}^{4\times 2}$ and $B \in \mathbb{R}^{2\times 3}$.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{bmatrix} \qquad B = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix}$$

We will consider partitioning the matrices into vectors that can be multiplied. We define

- The first row of A to be $a_1 = \begin{bmatrix} a_{1,1} & a_{1,2} \end{bmatrix}$.
- The second row of A to be $a_2 = \begin{bmatrix} a_{2,1} & a_{2,2} \end{bmatrix}$.
- The third row of A to be $a_3 = \begin{bmatrix} a_{3,1} & a_{3,2} \end{bmatrix}$.
- The fourth row of A to be $a_4 = \begin{bmatrix} a_{4,1} & a_{4,2} \end{bmatrix}$.
- The first column of B to be $b_1 = \begin{bmatrix} b_{1,1} & b_{2,1} \end{bmatrix}^T$.
- The second column of B to be $b_2 = \begin{bmatrix} b_{1,2} & b_{2,2} \end{bmatrix}^T$.
- The third column of B to be $b_3 = \begin{bmatrix} b_{1,3} & b_{2,3} \end{bmatrix}^T$.

then

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} \tag{A.4}$$

and so the product

$$\begin{aligned} AB &= \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} \\ &= \begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \\ a_4b_1 & a_4b_2 & a_4b_3 \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^2 a_{1,i}b_{i,1} & \sum_{i=1}^2 a_{1,i}b_{i,2} & \sum_{i=1}^2 a_{1,i}b_{i,3} \\ \sum_{i=1}^2 a_{2,i}b_{i,1} & \sum_{i=1}^2 a_{2,i}b_{i,2} & \sum_{i=1}^2 a_{2,i}b_{i,3} \\ \sum_{i=1}^2 a_{3,i}b_{i,1} & \sum_{i=1}^2 a_{3,i}b_{i,2} & \sum_{i=1}^2 a_{3,i}b_{i,3} \\ \sum_{i=1}^2 a_{4,i}b_{i,1} & \sum_{i=1}^2 a_{4,i}b_{i,2} & \sum_{i=1}^2 a_{4,i}b_{i,3} \end{bmatrix} \end{aligned}$$

By hand the easiest way to do this is to line up the two matrices to multiply as follows

$$\begin{array}{cc} & \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} \\ \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{bmatrix} & \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{bmatrix} \end{array}$$

Then calculate each component by lining up the row and column and multiplying the corresponding terms in them.

$$\begin{array}{cc} & \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} \\ \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{bmatrix} & \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & \square & \square \\ \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{bmatrix} \end{array}$$

Then repeat the process till every box is filled.

A.3 Scilab and Matlab Programming

Scilab is a Matlab look alike. We will use Scilab as it is free and open source, but it is useful to also know about Matlab.

A.3.1 Matlab

There are two basic ways to interact with Matlab: command line execution, and M-files. Yes there are others such as MEX-files, Simulink, and several interfacing programs, but they are not relevant to us. We will primarily be concerned with the use of M-files, because they are the most helpful. Command line execution is really just for quick operations and checking of segments of code. Matlab syntax is a high level programming language that interacts with a series of numerical libraries (most notably LinPack, EisPack, and BLAS). Like most programming languages we have two types of programs that can be written. A regular program, which is written as you would type commands on the command line, is the most basic type and is often the way you will start homework problems and other projects. Functions, which are sub-programs called by another program (even recursively by other functions), are probably the most useful, as they allow you to extend the language by defining new operations. One of the main goals of this class is for you to walk away with a library of Matlab functions that you can use to do a variety of tasks. So how do you specify which you want? You will get a regular program unless you start the M-file with the command function. The syntax is

```
function a=name(x,y,..., z)
```

or

function [a,b,...,c]=name(x,y,..., z)

The second form returns multiple values. Matlab gives us several command structures also: for, while, and if-elseif-else. To see how these work let's use the programs I passed out last time as an example.

Homework: Convert the Fortran program in 3.1 into Matlab syntax. Do problems 9, 13, 14 from section 3.1

Appendix B

Transform Techniques

Transform techniques covers a wide variety of mathematical methods. Probably the most used are the Laplace transform, the Fourier transforms, and the Z-transform. We will concentrate on the Laplace transform.

B.0.2 Laplace Transform

The two sided Laplace transform is defined by the integral transform

$$F(s) = \int_{-\infty}^{\infty} f(t)e^{-st}dt.$$

The one sided Laplace transform is defined by the integral transform

$$F(s) = \int_{0}^{\infty} f(t)e^{-st}dt.$$

Both have their reasons, but the one-sided transform is usually the one which is used. It has advantages of handling initial conditions, and some more subtle areas too detailed to go into right now. One key area to consider is the convergence of the integral. Does the integral even exist? For instance what if $f(t) = 1$ and $s = 0$? In this case we have the integral of 1 over an infinite length, and the integral does not converge (exist). In general s is restricted to the values in the complex plane, which will allow convergence. The question may arise as to the utility of the Laplace transform. Its main utility is in the ease of use, and the way it simplifies problems. Lets find some Laplace transform properties and then we will find some transform pairs and finally we will see how to use the Laplace transform on practical problems.

B.0.3 Properties

$$F(s) = \int_{0}^{\infty} f(t)e^{-st}dt$$

derivatives

$$\begin{aligned} F(s) &= \int_0^\infty f(t)' e^{-st} dt \\ &= (f(t)e^{-st})|_0^\infty + s\int_0^\infty f(t)e^{-st}dt \\ &= sF(s) - f(0) \end{aligned}$$

Convolution