NUMERICAL ANALYSIS NOTES

M. AUTH

1. Monday 28 January:

We will study, use, and analyze numerical algorithms to solve problems in algebra, linear algebra, calculus, and differnetial equations. Most of the material, besides some of the linear algebra, you will have already studied in your prior math courses. In those courses you were mostly concerned with showing (proving) that solutions to linear algebra, calculus, and differential equation problems have solutions. You did not worry about how to compute or even approximate these solutions. We will worry about these things in this class.

What we will see is that the techniques used to compute or approximate solutions in this course is often wildly different from the methods you used to prove that the solutions existed in previous math courses. One theme of the course that will appear over and over: Linear algebra is central to numerical analysis. Many problems are capable of being expressed in linear algebra terms and these expressions, although difficult to appreciate, are often the easiest way to approximate solutions to the problem numerically.

We covered Section 1.1 "The Bisection Method". We discussed how to bracket a root using Theorem 1.2. You should be able to state a rough proof of this theorem, i.e. you should at least be able to state the hypothesis and conclusion of this theorem. You need not understand the bisection code on page 26 but you should be able to generate the first three rows of the table on page 27 by hand.

Many problems we study in class will follow the template we use to study the "Bisection Method":

(1) Find an algorithm (think of a recipe to bake a cake) to generate a sequence of numbers $x_0, x_1, x_2, \ldots$.
(2) Use scrap paper to implement the algorithm on a simple problem.
(3) Code the algorithm into a computer language of your choice.
(4) Study when the sequence converges $\lim_{n\to\infty} x_n = x$.
(5) If the sequnence converges, determine how fast it converges.

The last step is most often the most challenging step, since we rarely know the value of x. After this last step we will always test our new algorithm by using it to approximate the solution to a simple problem that we already know the exact answer. These examples can confuse students.

Our algorithms are designed to be used to approximate solutions to problems that we cannot compute exactly. Yet, since this is a class, we always test out our algorithms and error formulas on problems we already know how to compute exactly from other math classes. This is the best way to learn the material. After we gain confidence in the algorithm by working through a few of these baby examples, we can safely try it on a problem that is unsolvable.

You should understand the solution error formula $= |x_c - r| < \frac{b-a}{2^{n+1}}$ for the bisection method and how to approximate a solution correct within p decimal places.

Here is the MATLAB bisection method code from our book:

```
% First attempt at bisection code

f=@(x) x^3 - x - 1;
a(1)=1; b(1)=2;
c(1) = (a(1)+b(1))/2;

for i = 1:13

    if f(a(i))*f(c(i)) < 0
        b(i+1) = c(i);
        a(i+1) = a(i);
    end
    if f(c(i))*f(b(i)) < 0
        a(i+1) = c(i);
        b(i+1) = b(i);
    end
    c(i+1) = (a(i+1)+b(i+1))/2;
end
n=1:14;
[n',a',c',b']
```

In order to understand this code, you must understand how assignment is done in computer laguages and how decisions are made in computer languages.

Here are some exercises to test your understanding of assignment. Try to work out what matlab will do in each of the following three code parts before you plug the code into your computer.

(1)
```
a = 2;
b = 3;
a = b;
disp(a)
disp(b)
```

(2)
```
a = 2;
b = 3;
b = a;
a
b
```

(3)
```
a = 2;
b = 3;
a = a + b;
b = a + b;
disp([b,a])
```

In order to generate a sequence of approximations, you will also need to understand how loops work in matlab. Each loop through the code will generate a new ("better") approximation. Like always try to figure out what is going on in the followin three loop by making a table on scrap paper before typing code into computer.

(1)
```
a = 0;
b = 0;
for i = 1:5
    a = a + 1;
    b = a + b;
    disp([a,b])
end
```
(2)
```
a = 0;
b = 1;
for i = 1:5
    a = a + b;
    b = a + b;
    disp([a,b])
end
```
(3)
```
a = 0;
b = 1;
for k = 1:5
    a = a + 1;
    b = a + b;
end
disp([a,b])
```

Lastly you need to understand how to make decisions with matlab. Try to determine what the following two code snipets produce before typing them into matlab.

(1)
```
a = 0;
b = 2;
if a>b
    disp(a+10)
else
    disp(a-13)
end
```
(2)
```
a = 0;
b = 2;
for i = 1:3
    if a>b
        a = a - 1;
    else
        b = b - 2;
    end
    disp([a,b])
end
```

To further review matlab, you can work through the matlab tutorial on our webpage.

2. Wednesday 30 January:

We studied the error formula $e_n = |x_n - r| < \frac{b-a}{2^{n+1}}$ for the Bisection method as well as the definition that an approximation x_n is correct within p decimal places if $e_n < 0.5 \times 10^{-p}$ in section 1.1 "The Bisection Method". If the hypothesis are met,

the Bisection Method always generates a sequence converging to a root. The error formula tells us how fast the sequence converges. Please read the section and do some of the assigned problems so that you become comfortable with approximations and error.

We finished class by introducing a different algorithm, Fixed Point Iteration (FPI). We use the FPI algorithm $x_{i+1} = g(x_i)$ to approximate fixed points $r = g(r)$ of the function g. If g is continuous and $\lim x_i = r$ then r is a fixed-point of g. We use this fact repeatedly. Unfortunately, many FPI sequences diverege—even when they start close to a fixed point. Please read this section and work through some of the problems. I will explain theorem 1.6 and the cobweb diagrams next week.

We studied Section 1.2 "Fixed-Point Iteration". Here is the FPI Matlab code I used in class to generate the first 19 elements in the FPI sequence. Please feel free to play around and alter my code. You'll learn a lot this way.

```
%FIXED_POINT Example 1.3 (p. 36) Fixed_Point Iteration
%Computes approximate solution to 0 = f(x) = cos(x) - sin(x)
% by finding a fixed point of x = g(x) = cos(x) - sin(x)
% We already know solution is pi/4.

g = @(x) x + cos(x) - sin(x);
x(1) = 0;  % intial guess

for i=1:19  % 19 iterations as book example
    x(i+1) = g(x(i)); % next approximation in the FPI sequence
end
x'  % display approximates in columns as in text.
```

In class we worked through some baby examples because the fixed-point could be found be hand by using basic high school algebra. Yet these high school problems are important and easy examples to practice your MATLAB code as well as your ability to make a few rows of a table like the one on page 32 by hand. We found that some fixed-point iterations converge and some are unstable. One good (but imprecise) way to see this geometrically is to make a cobweb diagram. You must be able to work with fixed point iterations in three ways: (1) by hand making tables and using basic algebra, (2) using MATLAB, and (3) by making a cobweb diagram.

Most importantly you must understand how to use the following theorem.

Theorem 2.1. *(theorem 1.6 from textbook) Assume that g is continuously differentiable, that $g(r) = r$ and that $S = |g'(r)| < 1$. Then Fixed-Point Iteration converges linearly with rate S to the point r for initial guesses close to r.*

After you understand the theorem work through many of the exercises and computer problems I've assigned from the end of the section. Doing exercises is the best way to learn the material (the answers to the odd problems are in the back of the book). Use theorem 1.6 to determine if FPI converges to a given fixed (with a close intial guess) and, if so, how fast it converges. We started to do this at the end of class.

3. Monday 4 January

Today we studied cobweb diagrams and used them to develop our intuition about if an FPI sequence converges or diverges. When we are given an FPI sequence $g(x)$ and x_0, it is helpful to think about it on many levels:

(1) Use basic algebra techniques from high school to solve $r = g(r)$ to find fixed points. (This only works for some baby problems that we do in class and on tests to learn more about FPI. In reality we never find the fixed point r; we use FPI to approximate it and that is the best we can do.)
(2) On scrap paper make a table to calculate a few $x_1, x_2, x_3, \ldots$ by hand and try to guess r for different initial guesses, x_0
(3) On scrap paper make a cobweb diagram for different initial guesses x_0, when possible.
(4) Use our Matlab FPI code to view some FPI sequences. Play around with the number of steps and the initial guess to view some different FPI sequences for the same $g(x)$.

After completing these steps, you can often understand the fixed points of $g(x)$ and the FPIs to converge to each fixed point given different initial guesses. Nevertheless, this analysis is not perfect. Some initial guesses x_0 give FPI sequences that diverge. Theorem 1.6 indicates that only initial guesses "sufficiently close" to r (with $|g'(r)| < 1$) lead to convergent FPI sequences. Sufficiently close can be a small interval or a large interval, depending on r and $g(x)$. If we study a different r or $g(x)$ we have to redo the analysis in the above steps. There are no shortcuts.

What's worse is that even if we complete the above steps (and we should), they are only approximations. If we really want to know that an FPI converges, and know how fast it converges, we need to prove it. This is a 300 level math course. You are going to have to prove some things. Proving that FPI converges often requires really understanding Theorem 1.6 and how the Mean Value Theorem of Calculus is used in the theorem. After that you will need to find bounds (maximums and minumuns) of $|g'(x)|$. You studied these things in your calculus course. It may be time to review.

4. Wednesday 6 February

In linear Convergence, we have the approximation $e_{i+1} \leq |g'(r)|e_i$ when r is a fixed point of the continuously differentiable function $g(x)$. It is a huge advantage to arrange that $S = |g'(r)|$ be small. For example if $S = .9$ then it takes about 22 iterates to stabilize a digit. While if $S = .1$ then it takes about 1 itenrates to stabilize a digit. Bisection: $S = 0.5$, FPI: $S \approx g'(r)$. But FPI may only converge locally, if at all?

Proving the Thereom 1.6 requires the Mean Value Theorem from calculus to prove the following theorem. After you understand the theorem by working through exercises and computer problems, please think a little about the proof.

5. Monday 11 February

Today we approximated a root of the equation $e^x + x = 7$. We know that the root r is between 1 and 2. In order to approximate it we turned this root equation into two FPI's $g_1(x) = 7 - e^x$ and $g_2(x) = \ln(7 - x)$. We could have also used the Bisection method but we wanted faster convergence so we tried FPI. After plugging

both g functions into matlab it was clear that g_1 gave a divergent FPI sequence while g_2 coverged quickly. In order to find the S factor in the g_2 FPI sequence we could not just compute $S = |g_2'(r)|$ since we do not know r. This is normal. Instead we had to use calculus to find a bound for $g_2'(c)$ for all c in the interval $[1, 2]$. These arguments are typical. You should know them.

We discussed the theorem 1.11.

Theorem 5.1. *(quadratic convergence theorem) Suppose that $g \in C^2$, and $r = g(r)$ and $g'(r) = 0$. Then for all iterates x_i sufficiently close to r, the errors $e_i = |x_i - r|$ satisfy the quadratic convergence estimate: $e_{i+1} \leq Me_i^2$ for some constant $M > 0$.*

You need not memorize the proof but here it is in any case.

Proof. Replace the MVT with first order Taylor expansion $g(x) = g(r) + g'(r)(x - r) + \frac{1}{2}g''(c)(x-r)^2$, where c is an unknown between x and r. This gives $|g(x) - r| = \frac{1}{2}|g''(c)|x - r|^2 \leq M|x - r|^2$. □

Theorem 5.2. *(theorem 1.11 from text): $f \in C^2, f(r) = 0, f'(r) \neq 0$ then Newton's iteration $g(x) = x - \frac{f(x)}{f'(x)}$ converges at quadratic rate if x_0 is sufficiently close to r.*

Proof. You need not memorize this proof but you should be able to use the quadratic convergence theorem above to prove this yourself. □

At this point we could spend some time finding an interval and M, a bound of $g''(x)$ on the interval. You will not be required to do this analysis of finding the intervals and inequalities for M in this class. Instead, we will use Matlab code

```
% NEWTONS METHOD: is used to make the approximation from Example
% 1.11 (p. 52) in our textbook

clear
f = @(x) x^3 + x - 1; % function to find roots.
fPrime = @(x) 3*x^2 + 1; % derivative of function.

x(1) = -0.7; % initial guess
for i = 1:7
    x(i+1) = x(i) - f(x(i))/fPrime(x(i));
end
x'  % displays approximates
```

When this converges, it will converge fast. You should see digits stabilize quickly. Keep the following fact in mind: Say x_0 has one decimal place accuracy.

- Linear Convergence (and $S = .1$) : It requires 49 iterations to obtain 50 decimal place accuracy.
- Quadratic Convergence (and $M = 1$) : It requires 6 iterations to obtain 64 decimal place accuracy.

6. Wednesday 13 February

In section 1.5 you only need to memorize the secant method. You need not know any other method in 1.5. You should learn to use matlab's *fzero* function. Please read section 1.3.

We covered 1.5 Root Finding Without Derivatives. It is only necessary to memorize the Secant Method from this section. It is good to know how to use Matlab's built in function *fzero* to check your approximate root, even though we do not know how fzero is implemented.

Here is the secant script I used in class

```
f=@(x) x^3 - 2*x - 2
x(1) = 1;
x(2) = 2;

for i =2:7
    x(i+1) = x(i) - (f(x(i))*(x(i) - x(i-1))) / (f(x(i)) - f(x(i-1)));
end
x'
```

Brent's Method is a hybrid method—it combines the property of guaranteed convergence from the Bisection Method, with the property of guaranteed convergence of more sophisticated methods. Matlab's *fzero* implements a version of Brent's Method, along with a pre-processing step, to discover a good initial bracketing interval–if one is not provided by the user.

Here is how I used *fzero* to do computer problem 2 in section 1.3

```
f=@(x) sin(x.^3) - x.^3;
fzero(f,0.1)
```

fzero gives an answer of -0.0014, only two decimal places of accuracy compared to the exact solution $x = 0$.

7. Wednesday 20 February

We quickly discussed the material from our hw sets in chapter 0. This material should be review. You should read it and do the problems on your own.

We also talked about section 1.3 Limits of Accuracy. One of the goals of numerical analysis is to compute answers within a specified level of accuracy. Working in double precision means that we store and operate on numbers that are kept to about 16 digits of accuracy. Can answers always be computed to 16 correct significant digits?

Example 7.1. 1.3 Limits of Accuracy

- Use Bisection to approximate root of $f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27}$ to within 6 correct significant digits.
- Exact Answer: $x = \frac{2}{3} = 0.666666\ldots$.
- The problem is that matlab thinks it has found a root $x = 0.666664123535156$.

Assume that f is a function and that r is a root. Assume that x_a is an approximate for r. The *backward error* of the approximation x_a is $|f(x_a)|$ and the *forward error* is $|x_a - r|$. The forward error is the only error we have studied before today. We used to call the forward error simply *the error*.

Now that we have two types of error to discuss, we can explain what happened in the previous problem. Bisection search was unable to approximate 6 significant digits of the solution because the backward error became much smaller than the forward error and the Bisection method stopped; this is called *the stopping criteria*.

The usage of "backward" and "forward" can be viewed by thinking of the "domain" of the problem (the function $f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27}$ in the case of $f(x) = 0$)

as the input of our method and the output as the solution ($x = 0.666664123535156$ in this case).

The subject of backward and forward error is relevant to stopping criteria for equation solvers. The goal is to find the root r satisfying $f(r) = 0$. How do we determine if x_a is a good approximation? Two possibilities: (1) "forward" make $|x_a - r|$ small or (2) "backward" make $|f(x_a)|$ small.

Whether forward or backward error is appropriate depends on the problem. For the Bisection method both errors are easy to bound.

Example 7.2. 1.3 Conditioning

- A problem is called sensitive if small errors in the input lead to large errors in the output.
- One way to measure sensitivity is the emf, the *error magnification factor*.
- The condition number of a problem is the maximum emf.
- Problems are then ill-conditioned or well-conditioned.

8. Monday 25 February

We covered sections 0.4 on "Loss of Significance" and section 1.3 on "Limits of Accuracy". It is important that you begin to understand the issues raised in these sections. Do not, however, fret over these problems. They are of lesser importance than the key algorithms and their analysis.

At the end of class we began covering section 2.1 on "Gaussian Elimination". I quickly reviewed Gaussian Elimination as you had seen it in your previous linear algebra course then I began to explain how our approach to Gaussian Elimination will be somewhat different in 328.

Please work through the "Matlab Tutorial" by Kelly Black on our webpage. In particular make sure that you play around with her Gaussian Elimination code in the "Loops" section.

9. Wednesday 27 Frebruary

We worked through sections 2.1 and 2.2 in our text. Please spend time solving linear systems $Ax = b$ by finding the LU and then carrying out two-step back substitution.

If you feel like you are in need of a linear algebra review, Gilbert Strang's video lectures $1, 2, 4, 5$, and 9 may serve as a helpful review. I have covered some of this material already in our class. There is a link to Professor Strang's vidoes on our coursepage.

10. Monday 4 March (Snowday)

I want you to read sections 2.3 and 2.4 in your textbook and try to do some of the assigned exercises and computer problems. Furthermore, I want you to watch the Prof. Strang's linear algebra lectures 4, 5, 6 (there is a link to these videos on our webpage). You will need to teach yourself about permutation matrices, forward error, and backward error in relation to the linear algebra equation Ax = b. In order to learn how matlab can be used to make these approximations, you should read the "Linear Equations" chapter on the Numerical Computing with Matlab link on our webpage http://math.sci.ccny.cuny.edu/pages?name=math328 Do not worry if you do not understand the code in section 2.7 of the "Linear Equation"

chapter and you do not need to read section 2.11 in the "Linear Equations" chapter. I want you to be comfortable using matlab's backslash and lu operators.

11. Wednesday 6 March

Exam 1.

12. Monday 11 March

We discussed sections 2.3 "Sources of Error". Please work through the problems in this section. There is a lot of terminology that you need to learn. You may need to reread this section several times throughout the semester.

I find it helpful to reread section 1.3 while reading section 2.3 because 1.3 discusses many of the same topics with respect of root approximations of solutions of $f(x) = 0$ instead of our favorite matrix equation $Ax = b$ of chapter 2.

13. Wednesday 13 March

We studied sections 2.5 "Iterative Methods"and section 2.4 "$PA = LU$ Factorization". You only need to know the Jacobi and the Gauss-Seidel methods from section 2.5 and the $PA = LU$ decomposition with partial pivoting from section 2.4.

In section 2.4 the matlab command

```
[L,U,P] = lu(A)
```

returns the P, L, and U matrices for a given square matrix A.

Here is the code I used in class for section 2.5 iterations to approximate a solution x to $Ax = b$. I will need to explain the Jacobi and Gauss-Siedel FPIs in class on Monday.

13.1. **First Iteration Code.**

```
% FIRST_FPI Solves linear system Ax=b using fpi x=Tx+c
% when T=I-A and c = b.

A=[3 1 -1; 1 -4 2; -2 -1 5];
T=eye(3)-A;
b=[3 -1 2]';
c=b;

x=[0 0 0]'; % initial guess
X=zeros(13,3); % FPIs

for i=1:12
    x= T*x + c;
    X(i+1,:)=x;
end
X
```

13.2. **Jacobi Code.**

```
% FIRST_JACOBI_FPI Solves linear system Ax=b using fpi x=Tx+c
% when Dx = -(L+U)x + b.  So T=-D^-1(L+U) and c=D^-1(b).

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

```
b=[3 -1 2]';  % initial problem

L=tril(A,-1);  % L = Lower triangular part (not LU decomposition)
U=triu(A,1);   % U = Upper triangular part (not LU decomposition)
D=diag(diag(A)); % D = diagonal part (not LDU decomposition)

T=-inv(D)*(L+U); % Jacobi iterate
c=inv(D)*b;

x=[0 0 0]'; % initial guess
X=zeros(13,3); % FPIs

for i=1:12
    x= T*x + c;
    X(i+1,:)=x;
end
X
```

13.3. **Gauss-Seidel.**

```
% GAUSS_SEIDEL computes first 7 iterates of Gauss-Seidel's method to
% solve system Ax=b when A is square b is column
% and x is initial guess.

L=tril(A,-1);  % L = Lower triangular part (not LU decomposition)
U=triu(A,1);   % U = Upper triangular part (not LU decomposition)
D=diag(diag(A)); % D = diagonal part (not LDU decomposition)

T=@(x) inv(L+D)*(b-U*x) % Gauss-Seidel iterate

for i = 1:7
    x = T(x)
end
```

14. Wedensday 20 March

We covered sections 2.6 "Methods for Symmetric Positive-Definite Matrices" and section 2.7 "Nonlinear Systems of Equations".

In section 2.6 you need only compute the Cholesky factorization by hand for 2×2 matrix, as we did in class. You need not program the Cholesky factorization. You are not responsible for subsection 2.6.4 "Preconditioning".

In section 2.7 you need only memorize and know how to use (by hand and as code) the "Multivariate Newton's Method". Here is my code for doing Newton's method:

```
% NEWTON27 Newton's method from example 2.32 in section 2.7
% Note that x = (x_1, x_2) is a vector.

f=@(x) [x(2)-x(1)^3, x(1)^2 + x(2)^2 - 1]' % function f
Jf=@(x) [-3*x(1)^2, 1; 2*x(1), 2*x(2)]; % Jacobian matrix of f
```

```
x=[1,2]' % initial guess

for i = 1:7
    x = x - inv(Jf(x))*f(x)
    end
```

It is not a good idea to use Matlab's *inv* command, since computing the inverse requires too much work. Our textbook illustrates a superior way to make this computation without computing the inverse.

15. Monday 25 March

We covered section 3.1. Interpolating polynomials are important in applications. The idea is that we approximate a potentially complicated function known by a table of its values with a polynomial. Polynomials are simple to work with so if our approximate polynomial is close the actual function we will understand much about the function.

There are two methods used to compute the Lagrange interpolation polynomial in section 3.1: 1. Using the functions $L_k(x)$ and 2. Using Newton's divided differences. You should know both methods.

Programming these methods on a computer is tricky. You can use the code in the book or my code.

```
% LAGRANGE_EVAL applies Lagrange interpolation to data points
% x, y, where x and y are n dimensional row vectors. The matrix C
% is the matrix of Newton divided difference coefficients. The
% Lagrange polyn is then evaluated at t.

C=zeros(n);  % square matrix to hold coefficients from Newton's
             % divided difference.
C(:,1)=y';   % first column of C is y

for j=2:n
    for i=1:(n-j+1)
        C(i,j)=(C(i+1,j-1) - C(i,j-1)) / (x(j+i-1) - x(i));
    end
end
% The first row of matrix C are the coefficients of the L.I.P.
% If you want to see these coefficients, include the line:
% disp(C(1,:))

% Now we evaluate the L.I.P. at the value t.
s=0; % evaluation
x_mult=1; % multiples of form (t-x1)(t-x2)...(t-xk)

for j = 1:n
    s = s + C(1,j)*x_mult;
    x_mult = x_mult*(t-x(j));
end
s
```

We discussed the Interpolation Theorem, Theorem 3.3, in section 3.2. I did not prove the theorem in class. You need not memorize the proof but you should know that the MVT played a large role. Focus your efforts on solving the hw problems like the example I did in class.

16. Wednesday 27 March

We covered section 3.4 "Cubic Splines". You are only required to understand the material describing how to turn the cubic spline problem with n data points into a linear system $Ax = b$ of $3n-3$ equations with $3n-3$ unknowns. This material can be found between p.166 through the bottom of p.169 in the second edition and between $p.173$ and the bottom of $p.176$ in the third edition.

We also covered section 4.1 "Least Squares". It is good to review orthogonality from linear algebra and calculus 3. One good way to do is is to watch Gilbert Strang's video lectures 14, 15 and 16. There is a link to these videos on our webpage.

17. Monday 1 April

We covered section 4.1 "Least Squares" and section 4.2 "A Survey of Models." Matlab commands used to solve the least square problem $A\bar{x} = b$ is

```
x = (A'*A) \ A'*b
```

We also talked about example 4.5 in section 4.1 where we use the normal equations to approximate 11 data points generated by the polynomial $y = f(x) = 1 + x+x^2+\ldots+x^7$ with a polynomial of the form $p(x) = c_0+c_1x+c_2x^2+\ldots+c_7x^7$. It is obvious that the solution to the normal equations should be $c_0 = c_1 = \ldots = c_7 = 1$. However the solution $c = (A' * A)$
$(A' * y)$ using MATLAB's system of equations solver to solve the normal equation $A^TAc = A^Ty$ is not $c = (1, 1, \ldots, 1)$. There is error. The error must be the result of the matrix (A^TA) being ill-conditioned. This is odd because the matrix A is not ill-conditioned. You should try this with matlab.

Since least squares problems are so important we need to find a better way to solve the normal equations $A^TA\bar{x} = A^Tb$ without magnifying error so much. This is done by factoring matrix A as $A = QR$. Since the matrix Q has orthogonal columns we will avoid having ill-conditioned problems resulting from "parallel" equations discussed in section 2.3

18. Wednesday 3 April

One way to achieve QR factorization is by using the Gram-Schmidt algorithm. This algorithm is important. You should know it. The key step in the algorithm is

$$\tilde{q_k} = a_k - (q_1^Ta_k)q_1 - (q_2^Ta_2)q_2 - \ldots - (q_{k-1}^Ta_k)q_{k-1}$$

where a_k is the $k-$th column of your starting matrix A and q_i is the $i-$th column of your orthogonal (columns) matrix Q which you generate one column at a time. I find it easiest to remember this step by thinking geometrically: For each column a_k subtract off all the parallel parts of previous columns to get $\tilde{q_k}$, a vector normal to all previous columns—but not yet unit length.

In order to program this into MATLAB (or Python, C++, ...), you must learn how to program the computer to add things up. For instance, here is how to tell MATLAB to compute $1 + 2^2 + 3^2 + \ldots + 71^2$:

```
sum = 0;
for i = 1:71
 sum = sum + i^2;
end
sum
```

You should understand this code before trying the Gram-Schmidt algorithm needed to get QR factorization of a matrix A. One such algorithm, *clgs*, is given in the section. You should understand it and use it—note that our author uses y instead of $\tilde{q_k}$ for obvious reasons in his code.

Householder reflectors is another way to compute the QR factorization. It will not be on the quiz on Monday—although it may appear on future exams. I will explain this technique after the quiz. You will never be quizzed or tested on the "full QR factorization."

19. Monday 8 April

We covered section 4.5 "QR Factorization". In particular we studied how to use QR factorization to solve a least square problem. We also learned a faster algorithm involving Householder reflectors to achieve QR factorization of a matrix. You should be able QR factorize using the Gram-Schmidt algorithm as well as Householder reflectors.

We covered section section 5.1 "Numerical Differentiation". Approximating derivatives is tricky business because it is easy to introduce error. We want to make h small in order to get a good approximate derivative but not too small that rounding errors enter our approximations. Please run the following code for Example 5.3 in section 5.1 :

```
% EXAMPLE53 In this code I tried to generate the table in
% example 5.3 on page 247 of section 5.1
% this is our 2 pt approx of f'(x).

% use this 2 pt approximate to approximate (e^x)' at x = 0.  We
% know the exat answer is f'(0) = 1.

for n = 1:9
    h = 10^(-n);
    d(n) = (exp(0+h) - exp(0)) / h; % d is a vector of approximate
    % derivatives for different h values.
    e(n) = abs(1 - d(n)); % error  computes the difference between
    % the know derivative of 1 and the d(n) approximate.
end
[d', e']  % matrix first column is approximates; 2nd column is errors.
```

One way to measure the error of a method is to discuss its order. For instance, the three-point centered difference formula is order 2. We can write it in the form $f'(x) = Q = F(h) + Kh^2$. It is a good practice using Taylor series to prove for yourself that it is indeed order two. Exercise 8 is also good practice using Taylor's theorem to verify order.

20. Wednesday 10 April

Extrapolation is a good way to reduce error (increase the order) of an algorithm without letting h get too small.

Please read section 5.2 "Newton-Cotes For Numerical Integration." In this section you will learn how to approximate integrals using a functions Lagrange Interpolation polynomial and error formula. In order to program these integration algorithms into a computer you must learn how to compute sums. Integration is one way to add things up.

We covered section 5.2 "Newton-Cotes Formulas for Numerical Integration". You should be able to derive the Trapezoid rule and its error form using your knowledge of Lagrange Interpolating polynomials. You need not be able to derive Simpon's rule or its error form. You should be able to easily turn the Trapezoid rule and Simpson's rule into their corresponding "composite" forms. You need not be able to derive the composite error formulas, though you should memorize (write on cheat index card) and understand how to use these error forms.

Here is my code to complete Example 5.8 in section 5.2:

```
% TRAPEZOID_METHOD Example 5.8 from section 5.1

f=@(x) log(x);

a = 1; b = 2;  % integral bounds

% Trapezoid method
m = 4;         % number of panels
h = (b - a)/m; % panel width
T = f(a) + f(b);  % T is the Trapezoid running sum
for i = 1:(m-1)
    T = T + 2*f(a + i*h);
end
T = (h/2)*T;
T
```

It is important that you really understand the above code and how it adds up all of the Riemann sums. This code together with our FPI code are the most important code in the course. You should also be able to code Simpson's rule for example 5.8.

```
% Simpson's method
m = 4;         % number of panels
h = (b - a)/(2*m); % panel width
S = f(a) + f(b);   % S is Simpson's running sum
% Add up the weights of 4
for i = 1:m
    S = S + 4*f(a + (2*i - 1)*h);
end
% Add up the weights of 2
for i = 1:(m-1)
    S = S + 2*f(a + 2*i*h);
end
S = (h/3)*S;
```

S

You should know this code well. You should play around with it to do different problems. Remember: the rough idea is to let h get smaller in order to get a better approximation. This only works to a point, however. When h gets too small (too many panels), computer errors can arise.

You will be required to use a computer on next week' (Wednesday 17 April) hw quiz. This will be our final hw quiz.

21. Monday 15 April

Learned how to derive the Trapezoid and Simpson Rules from the Lagrange Interpolation polynomial and error.

22. Wednesday 17 April

We covered section 5.3 "Romberg Integration". This is how extrapolation can be used on the Trapezoid Rule. As we saw in our example in class, extrapolation can improve the approximation significantly without forcing h to be too small.

Here is my Romberg code. Note there are two parts: (1) Find several Trapezoid approximations (each with half the previous stepsize h) and store them in the first column of your R matrix, and (2) Extrapolate this column knowing that the Trapezoid method is order two—finally we can continue to extrapolate one column at a time...

```
% ROMBERG_INTEGRATION
% Approximates the integral of f(x) on the interval a<=x<=b
% using Romberg integration in section 5.4 computer problem 2.
clear
f=@(x) exp(cos(x)); a=0; b=pi;  % function and boundaries

R=zeros(5); % Romberg Integration Tableau
R(1,1)=(b-a)/2*(f(a)+f(b));

% First approximate integral using Trapezoid rule with
% n=1,2,4,8,16 panels.
for i = 2:5
    h = (b-a)/2^(i-1);
    R(i,1) = 1/2*R(i-1,1);
    for k = 1:2^(i-2)
        R(i,1) = R(i,1) + h*f(a+(2*k-1)*h);
    end
end


% Extrapolate on first column of R
for j=2:5 % columns
    for i=j:5 % rows
        R(i,j) = (4^(j-1)*R(i,j-1) - R(i-1,j-1)) / (4^(j-1) - 1);
    end
end
```

R

Please read sectoin 6.1 "Initial Value Problems" over break. You need not worry about the Lipschitz constant or Lipschitz continuity. I find Arthur Mattuck's first three video lectures on differential equations (I've included a link to these videos on the bottom of our coursepage) contain a good review of differential equations for our purposes.

We covered section 6.1 "Initial Value Problems." You skip the material on the Lipschitz constant and Lipschitz continuity.

You should be able to calculate Euler approximations to IVPs by hand and using a computer. Here is my code for Euler's method:

```
clear

f=@(t,y) t^2 - y^2
t(1) = 0;
w(1) = 1;
h = 0.1;

for i=1:17
    t(i+1) = t(i) + h;
    w(i+1) = w(i) + f(t(i), w(i))*h;
end
[t',w']
```

23. Monday 29 April

Make sure that you can compute the exact answers to IVPs using separation of variables and first-order linear methods. Exercise 3 from 6.1 uses separation and exercise 4 from 6.1 uses first-order linear. If you have not taken math 391 at CCNY you will need to learn these two methods in order to complete these problems.

We covered sections 6.2 "Analysis of IVP solvers" and section 6.3 "Systems of ODEs". You need not understand the proofs in section 6.2. You should know that Euler's method is an order one method while the "Improved Euler's method" a.k.a the Trapezoid method is order two. You should be able to compute the Trapezoid method by hand and using code. Here is my code to compare Euler's method with the Trapezoid method and the exact answer to an IVP.

```
% COMPARE_METHODS
% Euler's vs. Improved Euler's vs. Exact Answer

clear
clf
f=@(t,y) y - 0.5*exp(t/2)*sin(5*t) + 5*exp(t/2)*cos(5*t); % ODE

h=0.01;  % stepsize.  Play around with different h values.
n=700; % number of steps to time t=7.  Play around with n.

% Euler's method to create piecewise linear approximation w(t) to
% the real solution y(t).
w(1)=0; % initial condition for euler's.  Try different intial values.
```

```
t(1)=0; % intial time
for i =1:n
    s1 = f(t(i),w(i));  % Euler's slope.
    w(i+1) = w(i) + h*s1;  % update w(t)
    t(i+1) = t(i) + h;  % update time.
end
plot(t,w,'r')  % Connect the dots of euler's approximate solution in red.
hold on;

% Improved Euler's method (or Trapezoid method or Heun method) is
% used to create piecewise linear approximation v(t) to
% the real solution y(t).
v(1)=0; % initial condition for euler's.  Try different intial values.
t(1)=0; % intial time
for i =1:n
    s1 = f(t(i),v(i));  % Euler's slope.
    s2 = f(t(i) + h,v(i) + s1*h);  % Euler's next slope. Sniff
                                   % ahead slope.
    v(i+1) = v(i) + h*(s1 + s2)/2;  % update w(t) with the average
                                    % of the Euler and Sniff ahead slopes
    t(i+1) = t(i) + h;  % update time.
end
plot(t,v,'k.')  % Connect the dots of euler's approximate solution in red.
hold on;

% Exact Solution.  We know from differential equations course.
t = 1:h:h*n; % time intervals
y = exp(t/2).*sin(5*t);  % y(t) is the exact solution
plot(t,y,'b')  % exact solution in blue
```

In section 6.3 we study problems where more than one variable is changing. Here is my rabbits and foxes code I used in class using the predator-prey model.

24. Wednesday 1 May

I started class doing some hw problems from sections 6.1 and 6.2 to compare Euler's method with the Trapezoid method and with the exact answer. We were particularly interested in what it meant that Euler's method is order 1 and the Trapezoid method is order 2.

Here is the code for the logistic model used to model the rabbit population on an island without any predators.

```
% EULERS_METHOD to model rabbit population using logistic model.

clear
clf
f=@(t,y) .1*y*(1 - y/10000);

h=0.1;  % stepsize in months
n=120*5; % 5 years into future
```

```
t(1)=0; % initial time
w(1)=8200; % initial number of rabbits

for i =1:n
    t(i+1) = t(i) + h; % update time
    w(i+1) = w(i) + f(t(i),w(i))*h; % update height, approximate solution
end
plot(t,w) % solution graph
hold on

t(1)=0; % initial time
w(1)=10900; % initial number of rabbits

for i =1:n
    t(i+1) = t(i) + h; % update time
    w(i+1) = w(i) + f(t(i),w(i))*h; % update height, approximate solution
end
plot(t,w,'b') % solution graph

t(1)=0; % initial time
w(1)=10000; % initial number of rabbits

for i =1:n
    t(i+1) = t(i) + h; % update time
    w(i+1) = w(i) + f(t(i),w(i))*h; % update height, approximate solution
end
plot(t,w,'r') % solution graph
```

Here is Euler's method used in the predator-prey model from class. This is our first example of a system of IVPs from section 6.3.

```
% EULERS_METHOD

clear
clf

% SET PARAMETERS FOR PREDATOR (FOX) AND PREY (RABBIT)
a=.1;  % 0.1 rabbit per month per rabbit growth
b=10000; % carrying capacity
c=0.005; % rabbit per month per rabbit-fox
d=0.00004; % fox per month per rabbit-fox
e=0.04; % fox per month

RPrime=@(R,F) a*R*(1 - R/b) - c*R*F;  % Change in Rabbits = R'
FPrime=@(R,F) d*R*F - e*F;  % Change in Foxes = F'

h=0.1;  % stepsize in months
n=120*20; % 20 years into future

t(1)=0; % initial time
```

```
r(1)=2000; % initial number of rabbits
f(1)=10; % initial number of foxes

for i =1:n
    t(i+1) = t(i) + h; % update time
    r(i+1) = r(i) + RPrime(r(i),f(i))*h; % update rabbit population
    f(i+1) = f(i) + FPrime(r(i),f(i))*h; % update fox population
end
plot(t,r,t,100*f) % solution graphs
```

We covered section 6.3 "Systems of Ordinary Differential Equations." We learned how to reduce a higher order differential equations to a system of first-order equations. This is important. Our approximate methods (Euler's, Trapezoid, RK4) only work for first order systems of equations.

Here is my pendulum code I used:

```
% EULERS_METHOD to approximate pendulum.

clear
clf

y1Prime=@(y1,y2) y2;  % Change in angle theta
y2Prime=@(y1,y2) -sin(y1);  % Change in angle prime (the derivative)

h=0.1;  % stepsize in radians
n=80*pi; % Go 8 radians forward

t(1)=0; % initial time
y1(1)=pi/3; % initial angle
y2(1)=0; % initial angular velocity

for i =1:n
    t(i+1) = t(i) + h; % update time
    y1(i+1) = y1(i) + y1Prime(y1(i),y2(i))*h; % update angle
    y2(i+1) = y2(i) + y2Prime(y1(i),y2(i))*h; % update angle velocity
end
plot(t,y1,t,y2) % solution graphs of angle and angle velocity
```

It is remarkable how the graphs of the $\theta(t), v(t)$ and solutions to the pendulum problem are similar to the rabbits and foxes from last class.

We also studied section 6.4 "Runge-Kutta Methods and Application". This is a favorite for those trying to quickly approximate a solution to an IVP. It requires computing a super-slope, a combination of four Euler-like slopes, in order to get to the next step in RK4. The extra computational effort is mostly worth it because the approximate is often much closer to the exact answer than other methods.

Here is my RK4 method:

```
% RK4 Implemented to solve Example 6.18

f=@(t,y) t*y +t^3  % D.E
y=@(t) 3*exp(0.5*t^2) - t^2 - 2; % exact solution from p. 283
```

```
% Compute RK4 for n = 5, 10, 20, 40, ..., 640 steps
a = 0; b = 1; % a = start time, b = end time
n = 5; % number of steps to start

while n <= 640 % loop for each n
    clear t; clear w; % clear variable for each n
    t(1) = 0; % initial time is 0
    w(1) = 1; % initial approximation is 1
    h = (b - a)/n; % stepsize

    for i = 1:n % loop to implement RK4
        s1 = f(t(i), w(i)); % the Euler slope
        s2 = f(t(i) + h/2, w(i) + h*s1/2); % sniff ahead slope
        s3 = f(t(i) + h/2, w(i) + h*s2/2); % another sniff ahead slope
        s4 = f(t(i) + h, w(i) + h*s3); % still another sniff ahead
        w(i+1) = w(i) + h*(s1 + 2*s2 + 2*s3 + s4)/6; % update w
        t(i+1) = t(i) + h; % update t
    end
    disp([n, h, w(n+1), y(1), abs(w(n+1) - y(1))]) % display (steps, h, approx,
                                        % error)
                                        %disp(abs(w(n+1)-y(1)));
    n = 2*n; % update n for next loop
end
```

For the exam on Wednesday we decided that each student could bring in a one page (ONE SIDED) formula sheet.

25. Monday 6 May

We reviewed for Wednesday's exam by doing exercises $1b$ and $2b$ in section 6.3. We then tried to run the computer code to do these exercises in computer problem 1 but my computer froze before we could see the code in action. Here is my code:

```
% My code for exercise 1b and computer problem 1 from section 6.3
% You should be able to alter my code to do 2b from section 6.3
y1Prime=@(y1,y2) -y1 - y2;
y2Prime=@(y1,y2) y1 -y2;

h = 0.25;

t(1) = 0;   % initials
y1(1) = 1;
y2(1) = 0;

for i = 1:9
    t(i+1) = t(i) + h;
    y1(i+1) = y1(i) + h*y1Prime(y1(i),y2(i));
    y2(i+1) = y2(i) + h*y2Prime(y1(i),y2(i));
end
[t',y1',y2']
```

We also covered sections 7.1 and section 7.2. In section 7.2 we only discussed 7.2.1 "Linear Boundary Problems". You are not responsible for section 7.2.2 "Nonlinear Boundary Problems" and I will not cover it in class.

26. Wednesday 8 May

Exam 2.

27. Monday 13 May

The last day of lecture today we covered sections 12.1, 12.2, and 12.3. Since we ran out of time, section 12.3 will appear on the final exam as a two point extra credit problem.

In section 12.1 you only need to understand the power iteration and inverse power iteration code. You need not understand the Rayleigh quotient iteration code.

Here is the code I used in class for the power iteration.

```
% Power Iteration: Program 12.1
clear

A = [1 2; 4 3];
x = [1 0]'; % initial guess

for k = 1:13   % I've chosen 13 steps of iteration.  You should change.
    u = x / norm(x);
    x = A*u;
    lam(k) = u'*x;
end
lam'
u=x/norm(x)
```

And here is my inverse power iteration code.

```
% Inverse Power Iteration: Program 12.1

A = [1 2; 4 3];
x = [1 0]'; % initial guess

for k = 1:13   % I've chosen 13 steps of iteration.  You should change.
    u = x / norm(x);
    x = A\u;
    lam(k) = u'*x;
end
lam'
u=x/norm(x)
```

Make sure that you know how to do these algorithms by hand as well and that you are not just typing into matlab.

In section 12.2 we only covered subsection 12.2.1. You may skip the rest of the section. Here is the code I used in class to illustrate the QR algorithm.

```
% QR_UNSHIFTED: as appears in section 12.2
```

```
A = [1 2; 2 1];
Q = eye(2,2);
Qbar = Q; R = A;

for k=1:15 % I've chosen 13 iterates.  Experiment by changing.
    [Q,R] = qr(R*Q);  % QR factorization from matlab
    Qbar = Qbar*Q;
end

lam = diag(R*Q)  % eigenvalues on diagonal
Qbar   % eigenvectors by repeated powers.
```

References