SFWR ENG 3X03

Teacher: Dr. Ned2

Fall 2013

[Numerical Methods Guy](https://www.youtube.com/user/numericalmethodsguy?feature=watch)

# Chapter 1

## Types of errors

The focus in this course is determining the error margin in MATLAB. We will only focus on discretization errors.

Assuming *u* is an exact result and *v* is an approximation

**Absolute error (A)**: |*u – v*|

**Relative error (R)**:

**Cancellation Error (R)**: when adding 2 numbers of similar absolute value, but opposite sign and there’s a loss of precision

**Overflow**: when a number is too large (fatal)

**Underflow**: when a number is too small (usually rounded down to 0)

: minimum value of *f*, with changing *x*

**Other sources**:

* **(A)** when y >> x, x + y
* **(A, R)** when |y| >> 1, xy
* **(A, R)** when |y| << 1, x/y

## Condition Number

**Forward error**: difference between result and solution

**Backward error**: the difference between the value of *x* used to find the result and the value of *x* that would give the solution

**Condition number**: how sensitive a function is to changes or errors in the input

Condition number = 

Condition number = 

Summary: condition number = change in *solution*/change in *input*

**Well-conditioned**: low condition #

**Ill-conditioned**: high condition #

**Horner’s Method (or rule or scheme)**: nested polynomial form, i.e.  


# Chapter 2 – Round-off Errors

## Floating Point

Floating Point (FP) System (*β*, *t*, *L*, *u*)

*β* – base

*t* – number of digits

*L* – lower bound for exponent

*u* – upper bound for exponent



Rounding unit (a.k.a. machine epsilon):

**flops**: the number of elementary floating point operations necessary to compute something; elementary floating point operations are considered equal to each other in terms of time and can be any of addition, subtraction, multiplication, and division

## Rounding

Types:

1. To nearest
2. Towards +∞ (i.e. upper bound)
3. Towards −∞ (i.e. lower bound)
4. Towards 0 / chopping

**Ceiling function**:

**Floor**:

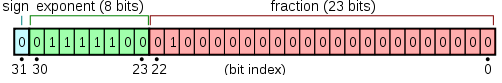
## IEEE Precision

+∞: from overflow of positives; maximum exponent, 0 fraction

−∞: from overflow of negatives; same as +∞, but with different sign

NaN: division by 0; maximum exponent; fraction is not 0

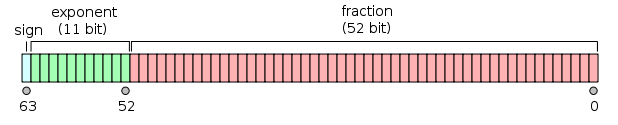
### Single

[](https://en.wikipedia.org/wiki/File:Float_example.svg)

A.K.A. short

Bias = 127

### Double

[](https://upload.wikimedia.org/wikipedia/commons/a/a9/IEEE_754_Double_Floating_Point_Format.svg)

A.K.A. long

Bias = 1023

## Convergence

x\*: root

More accurate as k →∞

**Rate of Convergence**: given, where 0 < *ρ* < 1, RoC =



## Taylor Series

Rewrite a function, f(x) as an infinite sum of other functions.



*a* is a constant that either you choose or it is given to you

It is easiest to do if you start off by finding the first couple derivatives and then plugging in *a*, so you can see how the function changes with each iteration.

### e.g.)

As you can see, it seems to have some sort of factorial coefficient

## Taylor’s Theorem

If you know the value of f(x0), but want the value of f(x0 + h):



Where ξ is some point between x0 and x0 + h.

# Chapter 3 – Solving Non-linear Equations

Find the roots.

**Tolerance**: a value that tells the computer what precision to stop computation

**Absolute Tolerance**: |xn − xn − 1| < atol

**Relative Tolerance**: |xn − xn − 1| < rtol∙|xn|

|f (xn)| < ftol ← tolerance based on function

### Bisection Method

When given an upper and lower estimate for a number, find the midpoint between the two points. If the midpoint is negative, it becomes the new lower limit. If it is positive, it becomes the new upper limit. Keep doing this until the end of time…

This is used on graphing calculators.

It is linearly convergent.

#### Error

Where c is the solution and cn is the computed solution at step n



### Newton’s Method

It runs faster than bisection method. When given a single guess (x0),



[How it works](https://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif):

1. Find the tangent line of the function at the point.
2. Find the point on the line that hits y = 0.
3. The point is your new x0. Go to step 1.

One problem with this method is that sometimes the computation gets into an infinite loop, going back and forth endlessly between 2 points. Thus, you require another field, **maximum iterations**, to prevent infinite loops.

If you don’t know the derivative of the function, you need to sub in the [secant formula](#_Secant_Method).

It is super-linearly convergent.

#### Error

.

### Secant Method





The order of convergence is the golden ratio,. It is super-linearly convergent

## Solving Decimal Exponents



# Chapter 4 – Linear Algebra Review

[Review eigenvalue/vector/matrix stuff](https://docs.google.com/file/d/0BxW61uJyyN8TbW1rWWlfSUppSjg/edit?usp=drive_web)

**Perturbation**: a change in a vector

= scalar if **x** is a vector and **A** is a matrix

**Algebraic Multiplicity**: the number of repeats of a root

**Geometric Multiplicity**: the number of different roots

If, then A is similar to B

**Vector norm**: regular definition of norm; 



A norm must have the following properties:

1. 
2. 
3. If

**Matrix norm**: same as vector norm, but for matrices , i.e. the sum of all the elements on the row of the matrix with the highest sum

**Euclidean norm**: base 2 norm; magnitude of something in Euclidean space

Two matrices are considered **orthogonal** to each other if 

Matrix multiplication review:

Recall

**Positive definite**: a matrix, **A**, is positive definite if the scalar, , is positive, as long as ****

**Residual**: 

**Spectral radius**: the eigenvalue of a matrix with the highest magnitude 

# Chapter 5 − Solving a Matrix

When trying to solve for *x*, where **A*x*** *=* **b**, where *A* is a real, non-singular (determinant ≠ 0), n×n matrix, and *b* is a real vector, there are two ways:

1. Direct methods (this chapter): exact solutions without [round-off errors](#_Chapter_2_–) through variations of [Gaussian Elimination](#_Gaussian_Elimination)
2. [Iterative Methods](#_Chapter_7): something like what is done for non-linear equations when direct methods fail

## Backward Substitution

To obtain an **upper triangular** matrix (elements below the diagonal are 0), make your matrix an upper triangular matrix

Cost:

## Forward Substitution

To obtain a **lower triangular** matrix (elements above diagonal are 0), make your matrix a lower triangular matrix through Gauss-Jordan row operations.

Cost: same as [backward substitution](#_Backward_Substitution)

## Gaussian Naïve Elimination

Use this to solve for your upper and lower triangular matrices of your size *n* square matrix.

First, find the upper triangular matrix:

Start at the top left element

1. Go from left to right:
   1. Compare the current value to the value underneath it to find the value of .
   2. Multiply each value of xi,j→n by .
   3. Repeat by comparing the current value to the value under the one you just compared it to, i.e. .
2. Repeat starting at the next column, next row, i.e. start each at xi=j

Make up your lower triangular matrix:

1. Your diagonal is all 1’s
2. Everything above the diagonal is 0’s
3. Use your *l* values from the U-matrix to fill in the last gaps



## Gaussian Elimination

A method of solving matrices that is useful for solving for *x*.

Cost: 

### e.g.)













## LU Decomposition

It is useful if you aren’t given *m* number of **b** vectors.

Do not switch rows!

L: lower triangular

U: upper triangular

If L is a lower triangular, non-singular matrix, its inverse is also lower triangular.

1. Use [Naïve Gaussian](#_Gaussian_Naïve_Elimination) to find the upper and lower triangular matrices.
2. *A = LU* => *L*(*U***x**) = **b**, which splits up into 
3. Solve *L***y** = **b** for **y**. ← O(n2)
4. Use **y** to solve the equation **y** = *U***x** for **x** ← O(n2)



Cost: O(n3) for the Gauss, but O(n2) for solving for **b**.

### Finding the Inverse of a Matrix

Given A, find B = A−1



### GE With Pivoting (GEPP)

Due to floating point arithmetic errors, you sometimes get something where 1 + small# = 1. To avoid this we **pivot** by switching some of the rows. You do this to avoid division by 0.

**Scaled GEPP**: multiplying a row by a constant

## Cholesky Decomposition

For symmetric, positive-definite matrices ONLY, you can represent the [lower triangular](#_Forward_Substitution) as the transpose of the upper triangular, so we can let *G* represent *L*, where you try to find *G*. This halves the required memory and operations required.



Matrix multiplication





O(n3)

# Chapter 6 – Linear Least Squares

**Data fitting**: forming a curve that fits between the points

[**Interpolation**](#_Chapter_10−Interpolation): drawing a curve along the points, guessing the shape

This method is a method of data-fitting

**Linear least square**: find a function that best fits the given data points. The function that you want is the one that minimizes the sum of the square of the distances from the point to the curve. Why squared?

Usually you’d be given the general shape of function.

Given *n* points, represent the distance between the points and the line by:

Equateto find the coefficients of *f* because that is when the function is at a minimum. To do this findand

## Matrices

If your system is Ax = b, and your given y equation is , represent your y equation by the Ax. Given a set of *n* points

|  |  |  |  |
| --- | --- | --- | --- |
| x | 1 | 2 | 3 |
| y | 2 | 3 | 5 |



Solve for x: x = b\A

Also, the ‘\’ sumbol is the equivalent of the MATLAB function, mldivide(A,b) or b\*inv(A).

# Chapter 8−Eigenvalue



λ = eigenvalue

x = eigenvector

There are 3 methods to compute eigenvalues:

1. [**Power method**](#_Power_Method): Finds the largest eigenvalue
2. [**Inverse Power method**](#_Inverse_Power_Method): Finds the smallest eigenvalue and accelerates
3. **QR Algorithm**: Finds all eigenvalues, but it’s difficult (graduate level).

## Power Method

Finds largest eigenvalue. Works well on large, sparse matrices.



Magnitude!

, but then you need to scale it by factoring out a number, such that one of the elements becomes 1

**Deflation**: once you have computed the largest eigenvalue, you can subtract it and repeat the method to find the other eigenvalues from highest to lowest

## Inverse Power Method

Finds smallest eigenvalue, instead of largest eigenvalue

Plug in A−1 instead of A into the [regular power method](#_Power_Method).

# Chapter 10−Interpolation

Interpolation is determining the value of a point based on the values of the points around it. You do not need to use all given points.

Multiple methods:

1. [Monomial basis functions](#_Monomial_Basis):
   1. Not useful: only for introducing the concept, yesterday
   2. Poorly conditioned
   3. O(n3) operations
   4. Mostly useful for theory
2. [Lagrange](#_Lagrange) (easier)
3. [Newton Basis functions](#_Newton_Basis_Functions)

## Monomial Basis

Given *n*−1 points, each row represents a different equation

**Vandermonde Matrix**:

e.g. (1,1),(2,3),(4,3)





### Error

blah

## Lagrange

Polynomial will be of order *n* for *n*+1 points



### Error

blah

## Newton Basis Functions

### Linear

Given you have to find the value on the curve at a certain point, find 2 points, x0 and x1, such that x0 is below the value and x1 is above the value.



Plug that back in and get: 

This can also be written as

### Polynomial

The order of your polynomial will be the 1 – the number of points you are given. The order of your polynomial will be the subscript of your function. For example, a quadratic equation would have the form:



To determine the constants, start by plugging in f(x0):



Then plug in x1



Finally, plug in x2



You can also write b2 as, where you put the number of points inside the square brackets. The order of the points doesn’t matter-the result will be the same, so b2 can also be re-written as:

Summary:

### Error

blah

# Chapter 15 – Integration Methods

There are 3 quadrature integration methods:

|  |  |  |
| --- | --- | --- |
| [Trapezoidal](#_Multiple_Segment_Trapezoidal)  File:Trapezoidal rule illustration.png | [Midpoint](#_Midpoint_Rule)  File:MidRiemann.png | [Simpson’s](#_Simpson’s_1/3_Rule)  File:Simpsons method illustration.png |

## Multiple Segment Trapezoidal Rule

Since this method finds the integration between a and b by finding the area assuming a straight line between f(a) and f(b), you have some error.



Error: 

To reduce the area, break it up into *n* segments, such that you have segments:, where

Since each segment is now a different trapezoid, instead of having one overall error, you have multiple errors, such that the first error is:



The textbook summed it up to: 

### Absolute Error

Notice how the *a* cancelled? Well that’s going to happen all the time, so you can simply assume that the error for a given segment, *i*, the error will be:



Therefore the total error:



However, the textbook sums it up to the following equation: 

## Midpoint Rule

A.K.A. Rectangle Method or Riemann sum



### Absolute Error



## Simpson’s 1/3 Rule

A type of quadrature that also [interpolates](#_Chapter_10−Interpolation) the line.

Choose 3 points, instead of 2, where the 3rd point is the midpoint between a and b. In order for the line to go through all 3 points, so it must be a curve => polynomial of order 2





Now [solve the matrix](#_Chapter_5_−) for the coefficients a0, a1, and a2.

Finally, solve the integral, where:





The textbook gives:

### Absolute Error



## Composite Methods

These methods are a version of the quadrature methods that involves splitting up the interval between the points *a* and *b*, into intervals of size *hi*. This *h* value is consistent between all the points, except in [adaptive quadrature](#_Adaptive_Quadrature).

## Adaptive Quadrature

There is another version of the quadrature methods that have a changing *h* value, such that there is a smaller *h* for steeper lines, called **adaptive quadrature**. The less adaptive the method is, the **stiffer**.

These methods have the same formulas as the other quadrature methods. However, they have a changing *h* value.

# Chapter 16−ODE’s

To solve for, you could use Euler’s formula:, where , but this is inaccurate, so use Runge

## Runge-Kutta

Uses Taylor Series



The second order Runge-Kutta only includes up to the part where you need to find the second derivative:

### e.g.)



However, you want to avoid having to find the f’, so they re-write it as:

, where:



With unknowns a1, a2, p1, and q11. However, there are 3 more equations determined that can help to determine the values.



Now the question is how much weighting you’re going to apply between the two slopes to solve for the unknowns. How this is done is by different guesses of the value of a2:

* [Heun’s Method](#_Heun’s_Method): 
* [Midpoint method](#_Midpoint_Method):
* [Ralston’s method](#_Ralston’s_Method):

### Heun’s Method

If you assume:,







### Midpoint Method

Assume





### Ralston’s Method

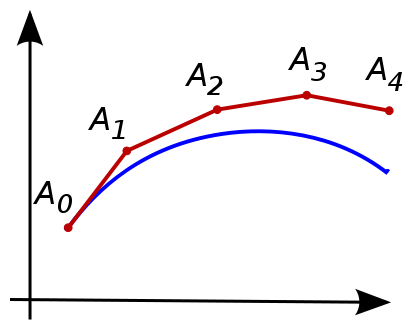




## Forwards Euler

This is an **explicit** method to find the solution for an ODE. Compared to its **implicit** counterpart, [Backwards Euler](#_Backwards_Euler), it is more efficient for less stiff equations

Use the tangent line to determine the next point of the line. The higher you set the step size (the less stiff), the more accurate.







### Error

blah

### Stability

The quality of **stability** is mainly to describe how the method responds to different step sizes (*h* values).

To demonstrate stability, use the **Dahlquist Test Equation**, which is where we assume

As an example, observe the analysis of the Dahlquist test equation using [forward euler](#_Forwards_Euler).



Notice how the power will appear to continuously increase? Stability is making sure the constant does not go to infinity as i approaches infinity. To do this, the initial value of the bracket must be between −1 and 1. To do this, set an *h* value such that:



## Backwards Euler

This implicit method for finding ODE’s is more efficient for stiffer equations than its explicit counterpart, [forwards Euler](#_Forwards_Euler).

This is the equation:

Since is on both sides of the equation, you need to bring it over to the left side and solve:

and solve for .

If , you would have:



If you take the limit as i → ∞, you can see that the fraction approaches 0, so this method is the more [stable](#_Stability) one.

### Error

blah