Part IA Numerical Methods Notes

- Fixed Point Representation

$$\ldots\ 2^2\ |2^1\ |2^0\ |\ .\ |2^{-1}|2^{-2}\ \ldots$$

- Overflow - When an operation on two representable numbers produces a result too small or large to represent
  - Will either wrap around or return smallest/ largest representable number
- Floating Point Representation
  - Numbers represented in the form $m \times 2^e$, where m is the mantissa and e is the exponent
  - Normally the base or radix is two, but can be other numbers
  - The number of bits used to store the mantissa is the precision
  - Multiplication/ Division

$$\left(m_0 \times 2^{e_0}\right) \times \left(m_1 \times 2^{e_1}\right) = (m_0 \times m_1) \times 2^{(e_0+e_1)}$$

$$\left(m_0 \times 2^{e_0}\right) / \left(m_1 \times 2^{e_1}\right) = (m_0/m_1) \times 2^{(e_0-e_1)}$$

    * The result may not be normalised
  - Addition/ Subtraction

$$\left(m_0 \times 2^{e_0}\right) \pm \left(m_0 \pm \left(m_1 \times 2^{e_1-e_0}\right)\right) \times 2^{e_0}$$

    * One must be shifted so that exponents are the same
    * May not be normalised

1 Numeric Base Conversions

- Integer Input
  - Input characters are processed in order
  - The previous running total is multiplied by ten and the new digit added on
- Integer output
  - Scan along the table, integer dividing by increasing magnitudes of 10
  - Each time output the character for that number
  - Then recurse on the remainder
- Floating Point Input
  - Identify the decimal point
  - Split into two, first integer part is the same as above
  - The fractional part is the same as above, start on the side away from the point and divide by ten each time
  - Add the two parts
  - Remember `E`, multiply by 10 to the power of what follows
- Floating Point Output
  - Choose a number of significant figures, s
  - Scale the number to be between 10^{s-1} and $10^s - 1$
  - Cast to an integer

  - Convert integer as normal, but add decimal point as appropriate

- Use the following code to round numbers

```
int round(double arg) {
    return floor(arg + 0.5);
}
```

2 IEEE Standard

- Two types, single and double precision

	Sign	Exponent	Mantissa
Single	1	8	23
Double	1	11	53

- The first digit in the mantissa is always 1 and therefore not represented, implied
  - Hidden bit: 24 significant bits with only 23 stored
  - Advantage of base 2
- Value represented is typically

$$(s?-1:1)\times 1.mmmmmm\times 2^{eeeeee}$$

- Two's compliment not used
  - Mantissa - Sign and magnitude representation
  - Exponent - Excess 127; $126\equiv -1, 127\equiv 0, 128\equiv 1$
- Excess 127 used because for positive numbers (excluding NaN) floating point comparison is the same as for integers
- Special Representations
  - 0 exponent and 0 mantissa = 0
  - 0 exponent and non-zero mantissa = NaN
  - 255 exponent and 0 mantissa = infinity
  - 255 exponent and non-zero mantissa = NaN
- Max positive around 3×10^{38}
- Min positive around 1.6×10^{-38}
- Numbers exactly representable in floating point are of the form

$$\pm((2^{23}+i)\ 2^{23})\times 2^{j}, \text{ where } 0\leq i<2^{23} \text{ and } -126\leq j\leq 127$$

- Require 9 significant figures to read it in exactly
- Have two representations of zero, positive and negative zero
  - differently signed zeros compare as equal
  - But

$$\frac{1.0}{0.0}=\texttt{inf} \qquad \frac{1.0}{-0.0}=\texttt{-inf}$$

- Overflow occurs when the exponent is too large to store

- Instead of throwing an exception return NaN or infinity
  - Will propagate rationally through computation and won't kill the system

$$\frac{0.0}{0.0} = \text{NaN}$$

- Compilers may not fully conform to standard, may pass more precise numbers to functions
- IEEE basic operations are defined as follows
  - Treat operands as exact
  - Round the result to nearest representable IEEE number
  - In the case of a tie choose the result with an even least significant bit
- IEEE rounding
  - Unbiased - Round to even rule: halfway case round to even
  - Towards zero
  - Towards positive infinity
  - Towards negative infinity
- A ULP is the unit in the last point
- Semi-monotonic - Is a function that is monotonic it should be monotonic
- Quantisation errors - arising from the inexact representation of constants in the program and read in data
- Rounding errors - produced by every IEEE operation
- Errors, for some a and an approximation b of a
  - Absolute Error

$$\text{Absolute Error is } \epsilon = |a - b|$$

  - Relation Error

$$\text{Relative Error is } \eta = \frac{|a - b|}{|a|}$$

- Represent a number as $\hat{x} = x \pm \epsilon$
- Errors in operations
  - $+$, $-$: sum absolute errors

$$(x_1 \pm \epsilon_x) + (y \pm \epsilon_y) = (x + y) \pm (\epsilon_x + \epsilon_y)$$

  - $*$, $/$: Sum the relative errors if small

$$(x(1 \pm \eta_x) * (y(1 \pm \eta_y)) = (x * y)(1 \pm (\eta_x + \eta_y) \pm \eta_x \eta_y)$$

    Discount $\eta_x \eta_y$ product as being negligible
- Because rounding random rounding errors average to 0
- Over a series of computations significance is gradually lost
- The machine epsilon is the difference between 1 and the least values greater than 1 that is representable in the given floating point type
  - For float 2^{-23}

  - For double 2^{-52}
- Machine epsilon gives an upper for the relative error of getting a floating point number wrong by 1 ULP
- The way errors build up or diminish is dependant on the sum
- If two numbers of different signs but similar magnitude are summed then the result will not be accurate
- When summing values
  - Sum starting from the smallest
  - Sum the smallest two elements and replace with their sum, until just one left
  - If some numbers are negative add them with positive numbers of a similar magnitude

3 Infinitary/ limiting computations

- Rounding Error - the error we get by using finite arithmetic during a computation
- Truncation Error - the error we get by stopping an infinitary process after a finite point
- Chebyshev Polynomials redistribute the error in a Taylor expansion across the input range
  - Functions are represented as a summation of the basis functions
- To calculate the truncation error sub in Taylor's expansion into the formula and cancel
- To calculate the rounding error use Taylor's expansion and assume results have error $\pm machups$
- Find optimum when rounding error equals truncation error
- In a first order algorithm halving the value to be optimised halves the truncation error
- In a second order algorithm halving the value to be optimised quarters the truncation error
- The bisection method
  - Form of successive approximation
  - Absolute error is halved each step and therefore has first-order convergence
  - Following pseudocode

```
Choose initial values a, b such that sign(f(a)) != sign(f(b));
Find mid point c = (a+b)/2;
If |f(c)| < desired_accuracy then stop;
If sign(f(c)) = sign(f(a)) then a = c else b = c;
goto 2
```

- Test is something will converge, test how it's error grows
  - Express ϵ_{n+1} in terms of ϵ_n
  - If the error decreases (eg. $\epsilon_{n+1} = k\epsilon_n$ for $k < 1$) then it will converge
- May enter an infinite loop called a limit cycle
  - May because there is no root (e.g. in Newton-Raphson)
  - May arise from discrete floating point values
- Newton-Raphson
  - For an equation $f(x) = 0$ Newton-Raphson will improve an initial estimate x_0

  - A second order convergence - for a precision n, requires $O(log\, n)$ steps

$$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)}$$

- Taylor's Series
  - How many terms, stopping early gives truncation error
  - Large cancelling intermediate terms can cause a loss of precision
  - Perform range reduction
    * Use identities to bring the argument into a small range
    * With large arguments reduction can be inexact, e.g. may require π to a very large accuracy
- Quadrature Techniques (numerical integration)
  - Mid-point rule - puts a horizontal line at each ordinate to mate rectangular strips
  - Trapezium rule - uses an appropriate gradient straight line through each ordinate
  - Simpson's rule - fits a quadratic at each ordinate
- Splining - Fitting a function to a region of data such that it smoothly joins up with the next region

3.1 CORDIC

- Find sine and cosine of Θ by rotating the unit vector

$$\begin{pmatrix} cos\,\Theta \\ sin\,\Theta \end{pmatrix} = \begin{pmatrix} cos\alpha_0 & -sin\,\alpha_0 \\ sin\alpha_0 & cos\,\alpha_0 \end{pmatrix} \begin{pmatrix} cos\alpha_1 & -sin\,\alpha_1 \\ sin\alpha_1 & cos\,\alpha_1 \end{pmatrix} \cdots \begin{pmatrix} cos\alpha_n & -sin\,\alpha_n \\ sin\alpha_n & cos\,\alpha_n \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\Theta = \sum_i \alpha_i$$

- Use the fact

$$cos\,\alpha = \frac{1}{\sqrt{1 + tan^2\,\alpha}}$$
$$sin\,\alpha = \frac{tan\,\alpha}{\sqrt{1 + tan^2\,\alpha}}$$

  to give

$$R_i = \frac{1}{\sqrt{1 + tan^2\,\alpha_i}} \begin{pmatrix} 1 & -tan\,\alpha_0 \\ tan\,\alpha_0 & 1 \end{pmatrix}$$

- Precompute values of $tan\,\alpha$, such that $tan\,\alpha$ is equal to a negative power of two
- Shift towards the axis, if too far go other way

4 Ill-conditionedness and Condition Number

- Ill-condtionedness - When a problem's solution is overly dependant on small variation in the inputs
- Found by taking the partial derivatives of the function around the point of interest
- The condition number for a numerical method is the worst amplification in relative error between input and output values

$$c_n = \frac{x}{f(x)} \cdot \frac{df(x)}{dx}$$

- Having a high condition number is ill-conditioned
- Backwards stability means that an inverse algorithm exists that can accurately regenerate the input
- backward stability usually implies well-conditionedness
- Can detect ill-condtionedness using Monte Carlo methods, calculate the function at a value and randomly alter the input by a small amount test how much the result changes

5 Solving Systems of Simultaneous Equations

- Use matrices, $Ax = b$
- Can use inverse, $A^{-1}Ax = A^{-1}b$, this can fail or be unstable
- Can use Gaussian Elimination - $O(n^3)$ complexity
  - If pivot is 0 or very small then will fail or have a large error respectively
- Better to use L/U decomposition
  - Perform Gaussian Elimination on A, to give a matrix in upper triangular form, this is U
  - The negation of the coefficient used to removed a given element is placed in L in the appropriate element

$$\begin{pmatrix} 1 & 5 & 5 \\ 2 & 3 & 3 \\ 4 & 7 & 4 \end{pmatrix}$$

  - To remove 2 the first row would have to be multiplied by -2 therefore 2 would be placed in L

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \sim & \sim & 1 \end{pmatrix}$$

  - The sub $A = LU$ into $Ax = b$ to give $LUx = b$
  - Next let $Ux = y$, sub this into $LUx = b$
  - This gives $Ly = b$ and $Ux = y$
  - Now solve $Ly = b$ for y then $Ux = y$ for x
- Can use Cholesky Decomposition if the matrix is symmetric positive definite
  - Split the matrix such that $A = LL^T$

$$L_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2}$$

$$L_{i,j} = \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) \text{ for } i > j$$

6 Alternatives to floating point

6.1 Interval Arithmetic

- Store an upper and lower bound
- Naturally copes with uncertainty and rounding does most of the work
- Is slower, some algorithms will converge but have huge bounds, if dividing and zero in input range then infinity in output range
- $x < y$ can be true and false

6.2 Arbitrary Precision Floating Point

- Lazily evaluated with significant figures
- If loss of significance then will redo previous calculation to higher precision
- Problem if result is zero as will keep evaluating trying to get significance

7 FDTD and Monte Carlo Simulation

- FDTD - finite difference time domain
- Event-driven - Simulation that uses discrete events and a time-sorted event queue
- Numeric, finite difference - Iterates over a fixed or adaptive time step
- Monte Carlo - Simulations that use random numbers
- Ergodic - After a certain number of iterations the system looses memory of its initial state
- The forward stencil or Euler's method - Use the rate values from the end of the previous timestep as though they are constant in the next timestep
- The backward stencil - Use the rate values at the end of the current timestep as though constant in this current timestep
- In FDTD simulations the state vector contains the variables need to be saved from one time step to the next
- The effectiveness of a simulation often depends on the size of the time step
- FDTD errors will cancel out if
  - They alternate polarity
  - They are part of a negative feedback loop that leads to equilibrium
- A non-linear component can be linearised
- The timestep can be dynamically changed for example

```
if N_it > 2N_max {ΔT *= 0.5; revert_timestep();}
else if N_it > N_max {ΔT *=0.9;}
else if N_it < N_min {ΔT *= 1.1;}
```

  where N_{it} is the number of interactions, N_{min} and N_{max} are the minimum and maximum number of iterations respectively, and ΔT is the timestep