Numerical Programming for Engineers

Dynamic Memory

- Don't know at compile time how much required
- Requested during runtime

Big O

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n)$$

$$O(n \log n) < O(n\sqrt{n}) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Linked Lists

- Struct that includes pointer to itself (the next node)
- Last item in chain has null pointer
- Can sequentially process by following the pointers
- Can push an element into the front of the chain
- Can pop an element off the chain
- Can add item to the tail of the list, if a tail pointer is kept (possibly in a list_t type)

Stacks and Queues

- Queue is FIFO, stack is LIFO

Priority Queues

- Items have a priority
- Operations
  - Enqueue (add item with priority)
  - Dequeue max (remove item with highest or lowest priority)
  - Find Max (find item with highest priority)
  - Change priority
- Applications in shortest path algorithms and bandwidth management
- Can be implemented via
  - Linked List
  - Array
  - Heap (Best)

BST

- Node has pointers to left and right
- Have smaller child elements on one side (left) and larger on the other (right)
- Can use polymorphic comparison and delete functions to allow for many data types
- Search is implemented recursively at each node

- Processing can be preorder, inorder, or postorder depending on when the data at the current node is processed relative to its subtrees

Heap

- A binary tree with a single key at each node, where the nodes are ordered vertically instead of left to right
- The key at each node is larger (or smaller) or equal to all nodes in left and right subtrees
- A max heap orders items in decreasing order, a min heap orders items in increasing order
- All levels are full except the lowest level, which is filled left to right
- Only requires an array to operate
- Construction O(n)
    - Insert all items into a binary tree in arbitrary order
    - Start with the rightmost node with at least one child, and fix heap properties by exchanging nodes
    - Repeat step 2 with the preceding non leaf node until the whole tree fulfils the heap properties
- Dequeue Max O(logn)
    - Swap max with the rightmost element
    - Remove the max (now the rightmost element)
    - Reheapify by sifting the new root down until it is in its rightful position
- Find Max O(1)
- Change Priority O(logn)
- Enqueue O(logn)

Heapsort

- Construct a heap in O(n) time
- Call DequeueMax n times to create the sorted array, at each operation the heap size is decreased by one
- Total cost is O(nlogn)
- Heap sort is not stable but is in place

Binary Search

- If array is sorted, can repeatedly half the array until the element is found
- Order of log(n)

Insertion Sort

- One part of the array is in sorted order (initially A[0])
- Increase the size of the sorted sub array by inserting items into the correct position
- Insertion is done by swapping elements into the correct position
- O(n^2) worst case

Merge Sort

- Divide and conquer approach
- Split array of size n into two arrays of size n/2
- Continue splitting recursively until size of 1

- Merge the sub arrays together and sort them whilst doing so
- Sort while merging by comparing first element of each array (as each subarray is sorted, first element will be largest/smallest)
- Merging is O(n)
- There are at most log2(n) levels
- Total complexity is O(nlogn)

Quick Sort

- Select an element as a pivot
- Partition the array so all elements with values less than the pivot are to the left, and all elements with values greater are to the right.
- Recursively apply this partitioning to sub arrays
- Put the sub arrays back together to get a solution for the initial array
- Partitions with two index counters coming from each end, increment the first index until its value is greater than pivot, then decrement the other index until its value is less than pivot, then swap. Repeat until two are on other sides of each other
- Total complexity is 1.44nlog(n) + O(n) which is just O(nlogn)

Pivots

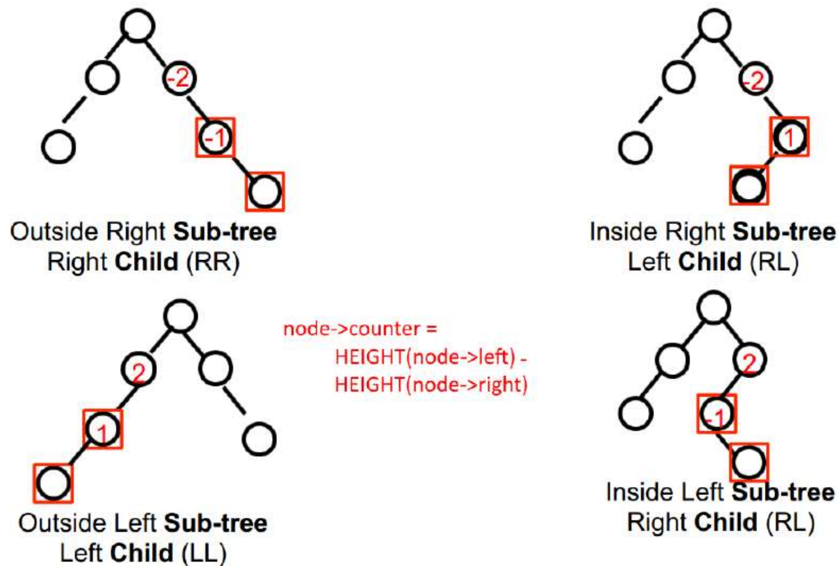
- Pivot selection is important, want to select one as close to the middle of the data as possible
- Taking the first element in a sorted array would result in a very poor pivot choice (more comparisons) and hence bad performance
- Safe options include taking the middle element, or even the median of three difference values. Best option would involve taking a random element
- Quicksort is not stable, meaning the original array order is not preserve d among items with equal sort key

Balanced BST

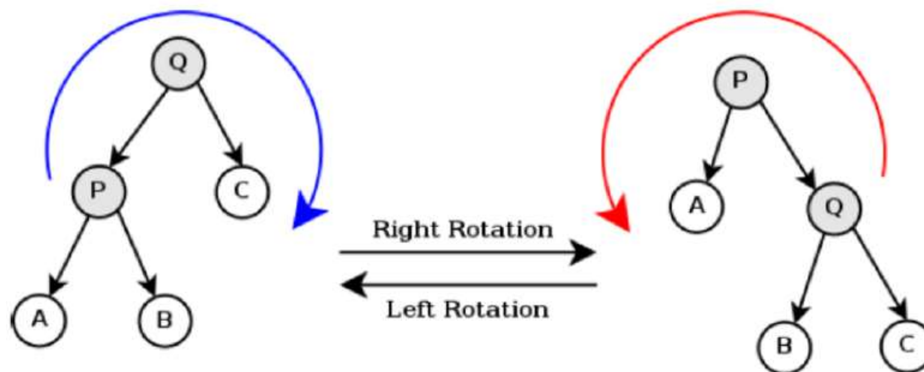
- Average case insertion and search if O(logn) for BST, worst case is O(n) for both, happens if binary search tree is just one long branch
- A balanced binary tree is desirable to attain O(logn) height
- Search in Balance BST is always O(logn), building is O(nlogn)

AVL Tree

- Keeps track of height of subtrees of every node
- Balances the node every time difference between subtrees is more than 1
- Types of unbalancing



- Can perform a rotation to balance the tree whilst still maintaining order, balancing adds time (but constant time)



2-3-4 Tree

- Can either contain 1 key/2 nodes, 2 keys/3 nodes, 3 keys/4 nodes
- Items go in between the nodes
- When 3 keys is reached and another key is added, the middle key is shifted up to the node above
- Items are only inserted into leaf nodes
- Tree grows in height slowly

Splay Tree

- Splay tree ensures the new item is always at the root of the BST
- Tree shape is rotated/modified depending on a zig-zig configuration or zig-zag on the item being searched for/inserted
- Frequently accessed items are higher up in the tree and can be accessed more efficiently

Time and Space Tradeoffs

- Adding a prev pointer to the node_t structure of a linked list

- Precompute and store frequently used sin() or cos() results
- Build a search index if lots of searches are performed over an input text
- Search for one pattern often: precompute a skip table for the pattern
- Search for many patterns in one text: build a search index

Lower Bounds

- Minimum amount of work needed to solve a problems
- e.g How many comparisons needed to sort an array of size n
- Considered tight if there exists an algorithm with the same efficiency as the lower bound
- Omega bounds the growth of a function from below, O bounds from above
-

Problem	Lower Bound	Tight?
Sorting	$\Omega(n \log n)$	Yes
Find Min	$\Omega(n)$	Yes
Search Sorted Array	$\Omega(\log n)$	Yes
$n \times n$ Matrix Multiplication	$\Omega(n^2)$	No

Subset Sum Problem

- Is there a subset of a set of numbers that adds up to a constant (i.e 1000)
- Given n integer values, and a target value k, need to evaluate all subsets of the n items which is 2^n complexity (exponential)
- Can check in constant time
- Is an example of a set problems that do not have known efficient algorithms to (polynomial time)

Numerical Limits

- Integers consume a fixed amount of storage space
- Unsigned Int
  - Maximum of (2^b)-1
  - Minimum of 0
  - Wraps around on overflow

Type	Smallest Value	Largest Value
uint8_t	0	255
uint16_t	0	65,535
uint32_t	0	4,294,967,295
uint64_t	0	18,446,744,073,709,551,615

- Signed Int
  - Maximum of (2^(b-1))-1
  - Minimum of 2^(b-1)

- One bit is used a sign bit

Type	Smallest Value	Largest Value
int8_t	-128	127
int16_t	$-32,768$	$32,767$
int32_t	$-2,147,483,648$	$2,147,483,647$

Floating Point

- Comprised of
  - The sign
  - Significand/Mantissa contains the actual digits of the number
  - Base and Exponent that describe where the decimal (or binary) point is placed relative to the beginning of the significand (for example 10^4 or 2^-20)
  - Positive exponents represent large numbers, whereas negative exponents are used to represent numbers close to zero

Sign	Significand	Base	Exponent	Scientific notation	Fixed point notation
$+$	5.23123	10	8	5.23123×10^8	523123000
$-$	1.234	10	-6	-1.234×10^{-6}	-0.000001234
$-$	6941.5	2	-12	-6941.5×2^{-12}	-0.2118378
$+$	11011101_2	2	-12	221×2^{-12}	0.006744385

Decimal number: 53127.4592

10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
5	3	1	2	7	4	5	9	2

$$
\begin{aligned}
53127.4592 = \; & 5 \times 10^4 \\
& +3 \times 10^3 \\
& +1 \times 10^2 \\
& +2 \times 10^1 \\
& +7 \times 10^0 \\
& +4 \times 10^{-1} \\
& +5 \times 10^{-2} \\
& +9 \times 10^{-3} \\
& +2 \times 10^{-4}
\end{aligned}
$$

Binary number: 11101.1001

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	1	0	1	1	0	0	1

$$
\begin{aligned}
11101.1001 = \; & 1 \times 2^4 \\
& +1 \times 2^3 \\
& +1 \times 2^2 \\
& +0 \times 2^1 \\
& +1 \times 2^0 \\
& +1 \times 2^{-1} \\
& +0 \times 2^{-2} \\
& +0 \times 2^{-3} \\
& +1 \times 2^{-4} \\
\\
& = 29.5625_{10}
\end{aligned}
$$

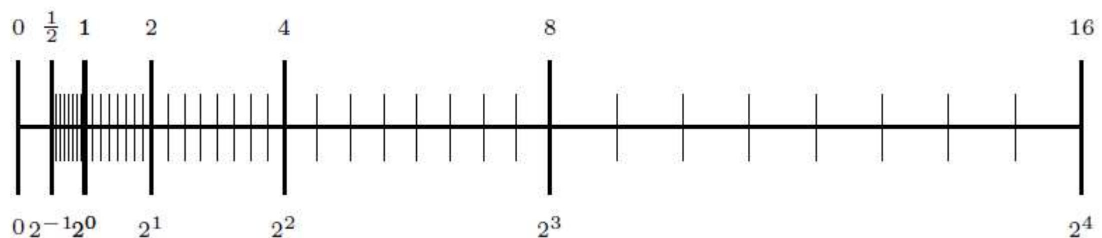
- IEEE 754 Standard
  - Standardizes the ways computers represent and operate on floating point numbers
  - Single Precision float
    - 32 bits total
    - 1 bit sign
    - 23 bit significand, stored in normalized form. Bit at position 2^0 is always set and is only stored implicitly, represents a value between 1.0 and 2.0. Largest binary number 1.99999988 = 2^0 + 2^-i (for i = 1 to 23)
    - 8 bits exponent (largest 127, smallest -126), stored as unsigned number with bias 127
    - All base 2
  - Single Precision Double
    - 64 bits total
    - 1 bit sign
    - 52 bit significand, stored in normalized form. Bit at position 2^0 is always set and is only stored implicitly, represents a value between 1.0 and 2.0
    - 11 bit exponent (largest 1023, smallest -1022), stored as unsigned number with bias 127
    - All base 2

Example: -763.56445:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
1	1	0	0	0	1	0	0	0	0	1	1	1	1	1	0
S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M

- There are limits to the usefulness of floating point, e.g for a number z =2147483008.00 the next larger number that can be represented with a float is 2147483136.00.
- The gaps between the numbers represented by a float depend on the magnitude of the number itself. Large number, large gaps; Small number, small gaps.
- Using a floating point that only has 3 bits of precision, between any of the powers of 2 there are only 2^3 = 8 representable numbers. However, the difference between two numbers changes as the exponent increases



Root Finding

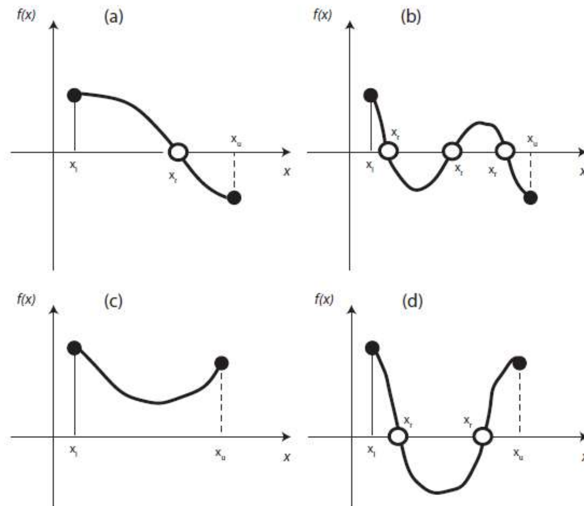
- Can find the roots and help solve ODEs

Graphical Methods

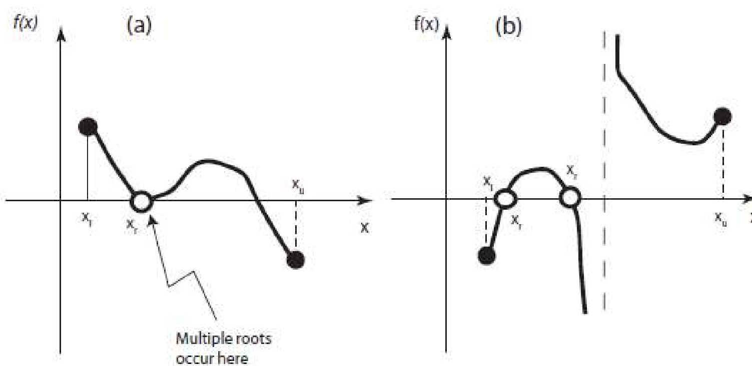
- Plot the graph and visually inspect where the function evaluates to zero
- Poor accuracy, and inefficient for solving for different values
- Advantages in seeing how the function behaves, quickly identifying number of roots, and as an initial guess for other methods

Bracketing Methods

- Called bracketing because they rely on having two initial guesses for an upper and lower bound.
- Guesses must bracket (be on each side of) the root
- This requirement can be hard to meet



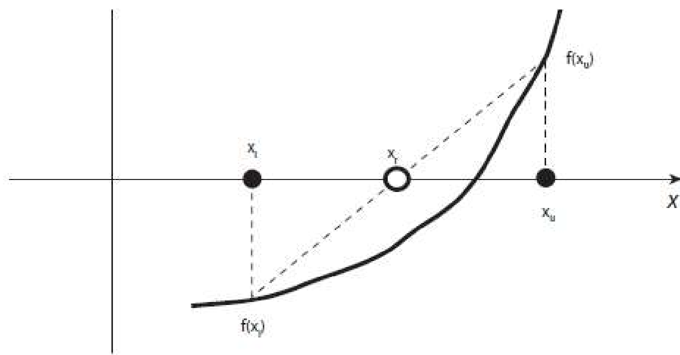
- (a)(b) If f(xu) and f(xl) are different signs, there must be at least one root, but there could be multiple (odd number)
- (c)(d) If they are the same sign, could be no roots or an even number of roots
- Exceptions occur when the function is tangential to the x axis or if there are discontinuities



Bisection Method

- Obtain two guesses xl and xu that bracket the root xr
- In general if the function values f(xu)*f(xl) < 0 you can be reasonably sure you have at least one root in between the two guesses
- Estimate the root of f(x) to be
- $x_r' = \frac{x_l + x_u}{2}$
- Now
  - If f(x'r) = 0, you just found the root
  - If f(xl)*f(x'r) < 0, then your root is in the lower subinterval, set xu = xr and repeat
  - If f(xu)*f(x'r) < 0, then your root is in the upper subinterval, set xl = xr and repeat
  - Usually converges to within an acceptable bound of 0 (epsilon)
- Oscillates towards the solution

False Position



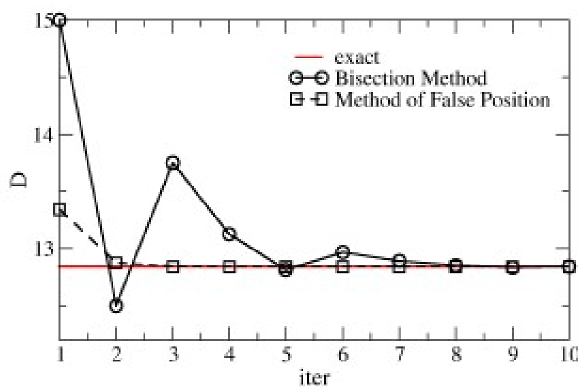
- If f(xl) is closer to zero than f(xu) the root is likely to be closer to xl than xu. Using similar triangles, we can obtain the following expression

$$\frac{-f(x_l)}{x_r - x_l} = \frac{f(x_u)}{x_u - x_r}$$

- And therefore

$$\boxed{x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}}$$

- Use the bisection method but replace second step with the false position formula to estimate a root location in between the two (using similar triangles)
- Monotonically converges to the solution, much faster than bisection method



Open Methods

- Only require either one initial guess or two initial guesses which do not have to bracket the actual root
- In general if they converge, it is much faster than bracketing methods

- However, unlike bracketing methods which always converge, they may not converge

Fixed Point iteration

- Rearrange the function f(x) to get x on one side by itself

$$f(x) = e^{-x} - x = 0.$$

$$x = e^{-x}$$

$$x_{i+1} = e^{-x_i}$$

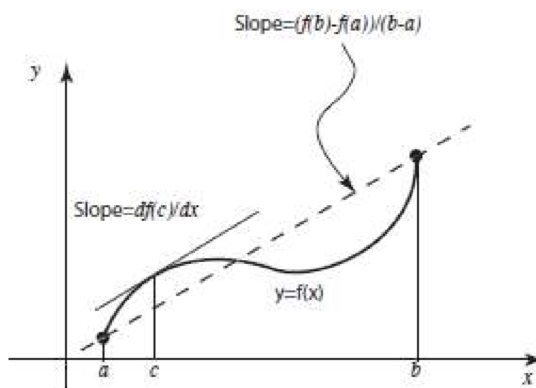
$$\boxed{x_{i+1} = g(x_i)}$$

- Now you have a function g(xi) which you continue to iterate through to the find the next xi
- Iterate until

$$|g(x_i) - x_i| < \varepsilon$$

- If f is a continuous function and differentiable between (a,b), then a number must exist between a and b such that the derivative of that function g is

$$g(b) - g(a)/(b - a)$$

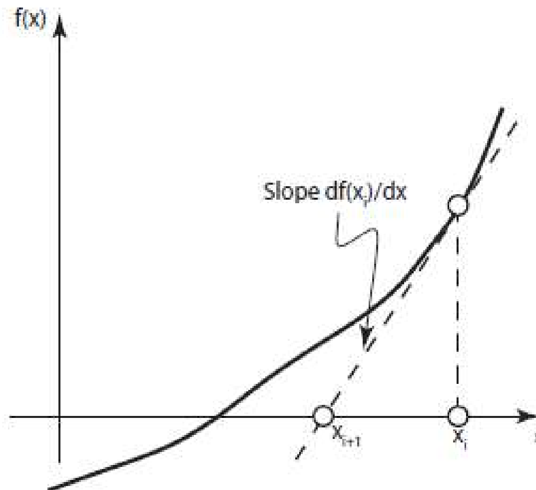


- Hence the condition for convergence is that for the newer guess to be closer to the root than the older guess is that

$$|dg(\xi)/dx| < 1.$$

Newton-Raphson

- Most widely used
- Converges quickly with one initial guess



- Expresses a guess for the root in terms of the derivative of the guess

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

- Let the initial guess be xi
- Find xi+1 using the equation
- Repeat with xi = xi+1 until the answer is accurate enough
- The magnitude of error can be proved to be roughly proportional to the square of the previous error
- Therefore convergence is quadratic (only if the derivative at the root is not equal to zero)
- Pitfalls
  - Requires evaluation of derivative which can be difficult
  - Has problems overcoming local maxima/minima

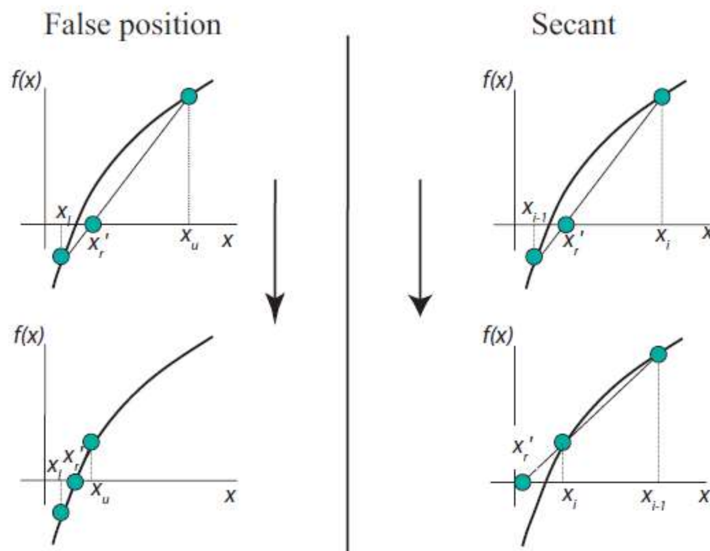
Secant Method

- Similar to Newton-Raphson method, but derivative is approximated by the backward finite difference

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

$$x_{i+1} = x_i - \frac{(x_i - x_{i-1})\, f(x_i)}{(f(x_i) - f(x_{i-1}))}$$

- Requires two estimates of xi and xi-1 to start, but is not classified as bracketing because function is not required to change signs



Root finding for System of nonlinear equations

Systems of Linear Algebraic Equations

- Can be represented in matrix form [A]{X} = {C}
- Can be solved by inverting the matrix A and multiplying {C} by inv[A], but difficult if [A] is large.
  - Compute [A]^-1: 2n^3 operations
  - Multiply {X} = [A]^-1{C}: 2n^2 operations

Direct Methods

- No iterations are involved

Gauss Elimination

$$\left[\begin{array}{cccccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & . & . & . & . & a_{1n}^{(1)} & a_{1,n+1}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & . & . & . & . & a_{2n}^{(1)} & a_{2,n+1}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & . & . & . & . & a_{3n}^{(1)} & a_{3,n+1}^{(1)} \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ a_{n1}^{(1)} & a_{n2}^{(1)} & a_{n3}^{(1)} & . & . & . & . & a_{nn}^{(1)} & a_{n,n+1}^{(1)} \end{array}\right]$$

- Augment [A] with {C} and then forward eliminate to get the matrix into solvable form

$$\left[\begin{array}{cccccccc|c} a_{11}^{(n)} & a_{12}^{(n)} & a_{13}^{(n)} & a_{14}^{(n)} & . & . & . & a_{1n}^{(n)} & a_{1,n+1}^{(n)} \\ 0 & a_{22}^{(n)} & a_{23}^{(n)} & a_{24}^{(n)} & . & . & . & a_{2n}^{(n)} & a_{2,n+1}^{(n)} \\ 0 & 0 & a_{33}^{(n)} & a_{34}^{(n)} & . & . & . & a_{3n}^{(n)} & a_{3,n+1}^{(n)} \\ 0 & 0 & 0 & a_{44}^{(n)} & . & . & . & a_{4n}^{(n)} & a_{4,n+1}^{(n)} \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & . & . & . & . & a_{nn}^{(n)} & a_{n,n+1}^{(n)} \end{array}\right]$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} \quad , \quad a_{ij}^{(k)} = 0 \quad , \quad a_{ij}^{(k)} = a_{ij}^{(k-1)} - \left(\frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} \right) a_{k-1,j}^{(k-1)}$$

- Solve with back substitution

$$x_n = \frac{a_{n,n+1}^{(n)}}{a_{nn}^{(n)}} \text{ and } \qquad x_i = \frac{a_{i,n+1}^{(n)} - \sum_{j=i+1}^{n} a_{ij}^{(n)} x_j}{a_{ii}^{(n)}}$$

for $i = n-1, n-2, n-3, .., 2, 1.$

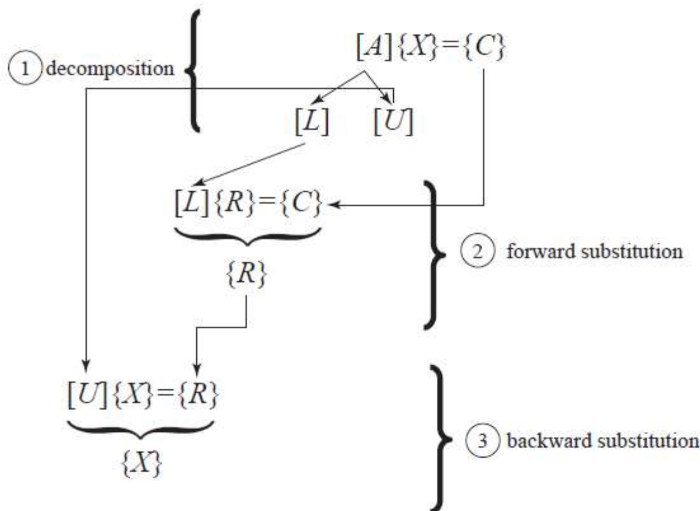
- Pitfalls
  - Very expensive
  - Formula for forward elimination requires division by diagonal components of [A], need to rearrange equations to make sure there is no division by zero (called partial pivoting) and also avoid dividing by small numbers

LU Decomposition

- In solving [A]{X} = {C}, if [A] is a 'full' matrix, it takes many steps to solve it.
- Would be preferable to express [A] as [L][U] where [L] and [U] are lower and upper triangular matrices.

$$[L] = \begin{bmatrix} l_{11} & 0 & 0 & 0 & . & . & 0 \\ l_{21} & l_{22} & 0 & 0 & . & . & 0 \\ l_{31} & l_{32} & l_{33} & 0 & . & . & 0 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ l_{n1} & l_{n2} & l_{n3} & l_{n4} & . & . & l_{nn} \end{bmatrix} \quad [U] = \begin{bmatrix} 1 & u_{12} & u_{12} & u_{12} & . & . & u_{1n} \\ 0 & 1 & u_{23} & u_{24} & . & . & u_{2n} \\ 0 & 0 & 1 & u_{34} & . & . & u_{3n} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & . & 1 \end{bmatrix}$$

- Therefore
  [L][U]{X} = C
  [U]{X}={R}
  [L]{R}={C}
- Now the original problem has been replaced with
  [L]{R}={C}
  [U]{X}={R}
  Which is much easier to solve because [L] and [U] are both lower/upper triangular matrices and only backwards/forward substitution is required
- Steps
  - Decompose [A] into [L] and [U]
  - Solve [L]{R} = {C} and obtain {R} by forward substitution
  - Use {R} obtained in above step and substitute into [U]{X}={R} so that you can solve for {X} by backward substitution



- To find [L] and [R]

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Through matrix multiplication, we know that

$$l_{11} = a_{11}, \qquad l_{21} = a_{21}, \qquad l_{31} = a_{31}, \qquad l_{41} = a_{41}.$$

$$l_{11}u_{12} = a_{12}, \qquad l_{11}u_{13} = a_{13}, \qquad l_{11}u_{14} = a_{14}.$$

$$u_{12} = \frac{a_{12}}{l_{11}}, u_{13} = \frac{a_{13}}{l_{11}}, u_{14} = \frac{a_{14}}{l_{11}}$$

- Keep alternating between getting a column of [L] and a row of [U], in general the formula is called Crout's LU decomposition algorithm

$$\begin{aligned}
& l_{i1} = a_{i1} \quad \text{for} \quad i = 1, 2, \ldots, n \\
& u_{1j} = \frac{a_{1j}}{l_{11}} \quad \text{for} \quad j = 2, 3, \ldots, n \\
& \\
& \text{Begin} \quad \text{for} \quad \text{loop} \quad j = 2, 3, \ldots, n \\
& l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} \qquad \text{for} \quad i = j, j+1, \ldots, n \\
& u_{jk} = \frac{a_{jk} - \sum_{i=1}^{j-1} l_{ji}u_{ik}}{l_{jj}} \qquad \text{for} \quad k = j+1, j+2, \ldots, n \\
& \text{End} \quad \text{for} \quad \text{loop} \\
& \\
& u_{ii} = 1.0 \quad \text{for} \quad i = 1, 2, \ldots, n
\end{aligned}$$

- After obtaining [L] and [U], solve for {R} by forward substitution

$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{Bmatrix} = \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{Bmatrix}$$

$$r_1 = c_1/l_{11}$$
$$r_i = \frac{\left(c_i - \sum_{j=1}^{i-1} l_{ij} r_j\right)}{l_{ii}} \quad \text{for} \quad i = 2, 3, 4, \ldots\ldots, n$$

- Finally solve [U]{X} = {R} with back substitution

$$\begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{Bmatrix}$$

$$x_n = r_n$$
$$x_i = r_i - \sum_{j=i+1}^{n} u_{ij} x_j \quad \text{for} \quad i = n-1, n-2, n-3, \ldots\ldots, 2, 1$$

- Alternatively to Crout's method, can factorise the matrix by having ones on the diagonal of the [L] matrix instead. This is called Doolittle's method
- Cost
  - Decompose [A] into [L][U]: 2/3n^3 operations
  - Solve [L]{R} = {C}: n^2 operations
  - Solve [U]{X} = {R}: n^2 operations
- If [A] becomes very large, the cost of direct methods can be bad as they scale with n^3

Iterative Methods

- $$[M]\{X\}^{(k+1)} = [N]\{X\}^{(k)} + \{C\}$$
- Where {X}^k is the current (kth) guess of the true solution {X}
- Want to choose [M] so that it is easier to solve than the original express [A]{X} = {C}
- General procedure
  - Define [M] and [N]
  - Obtain initial guess {X}0
  - Solve the iteration for a new guess
  - Define and obtain maximum absolute value of residual vector {r}
  - $$\{r\} = [A]\{X\}^{(k)} - \{C\}$$
  - End when residual value is less than epsilon

Point Jacobi Method

- [M] consists of only the diagonal elements of [A]

- The negative of all other elements is placed into [N]

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix}$$

$$[M] = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} \quad , \quad [N] = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}^{(k+1)} = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}^{(k)} + \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix}$$

- Therefore

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(\sum_{j=1 j \neq i}^{N} -a_{ij} x_j^{(k)} + c_i \right)$$

Gauss-Seidel Method

- Most commonly used iterative method
- [M] consists of the lower triangular elements of [A]

$$[M] = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} [N] = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}^{(k+1)} = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}^{(k)} + \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix}$$

- Solve the system from the top row to the bottom row gives you three equations for xi^(k+1)
- Make sure none of the aii values are zero
- Need initial guesses for x1(0), x2(0), x3(0)
- Substitute guess values for x2(0) and x3(0) into the equation for the first row to get a better estimate for x1, call this x1(1)
- Substitute x1(1) and x3(0) into the second row to get a better estimate for x2, called x2(1)

- Use x1(1) and x2(1) to get a better estimate for x3 called x3(1)

$$\left.\begin{aligned} x_1 &= \frac{c_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 &= \frac{c_2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 &= \frac{c_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \end{aligned}\right\} \text{First iteration}$$

$$\left.\begin{aligned} x_1 &= \frac{c_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 &= \frac{c_2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 &= \frac{c_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \end{aligned}\right\} \text{Second iteration}$$

- Generic formula for xi(k+1) for an N*N matrix is

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(-\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{N} a_{ij}x_j^{(k)} + c_i\right)$$

- Convergence criteria
  - Defining an error matrix at the kth iteration to be

    $$\{\epsilon\}^{(k)} = \{X\}^{(k)} - \{X\}$$

  - We can find an expression for epsilon^k by subtracting the exact solution from the approximation and find that it is related to initial error

    $$\{\epsilon\}^{(k+1)} = [P]^{k+1}\{\epsilon\}^{(0)}$$

  - If we diagonalize [P] with matrix S (where columns of S are eigenvectors of P)

    $$\{\epsilon\}^{(k+1)} = [S][\Lambda]^{k+1}[S]^{-1}\{\epsilon\}^{(0)}$$

  - Hence for error to go to zero as fast as possible, magnitude of all of the eigenvalues of [P] should be less than 1

Least Square Approximations

- How to draw a line of best fit through data

- Represent as y = ax + b, where values of a and b are so that the line is closest to the actual values
- Error can be represented as

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{(ax_i + b)}_{\text{predicted value}}$$

- With N data points, square the error at every point and sum it to find total error

$$S = \sum_{i=1}^{N} (y_i - ax_i - b)^2$$

- To find the minimum value of error, compute dS/da and dS/db and set it to zero (hold xi and yi as constants while differentiating as they are data points)

$$\frac{\partial S}{\partial a} = \sum_{i=1}^{N} 2(y_i - ax_i - b)(-x_i) = 0$$

$$\frac{\partial S}{\partial b} = \sum_{i=1}^{N} 2(y_i - ax_i - b)(-1) = 0$$

$$a\sum_{i=1}^{N} x_i^2 + b\sum_{i=1}^{N} x_i = \sum_{i=1}^{N} x_i y_i$$

$$a\sum_{i=1}^{N} x_i + bN = \sum_{i=1}^{N} y_i$$

- Solve it as a system of 2x2 linear equations

$$\begin{bmatrix} \sum_{i=1}^{N} x_i^2 & \sum_{i=1}^{N} x_i \\ \sum_{i=1}^{N} x_i & N \end{bmatrix} \begin{Bmatrix} a \\ b \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^{N} x_i y_i \\ \sum_{i=1}^{N} y_i \end{Bmatrix}$$

Polynomial Least Squares Approximation

- Assume a relationship of the form

$$y = a_0 + a_1 x + a_2 x^2 + \ldots\ldots + a_n x^n$$

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{\left(a_0 + a_1 x_i + a_2 x_i^2 + \ldots\ldots + a_n x_i^n\right)}_{\text{predicted value}}$$

- Similar to linear least squares, set dS/dai to zero (all the derivatives with respect to the different a values)

$$\begin{bmatrix} N & \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & \sum x_i^n \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \cdots & \sum x_i^{n+1} \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \cdots & \sum x_i^{n+2} \\ \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \sum x_i^6 & \cdots & \sum x_i^{n+3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sum x_i^n & \sum x_i^{n+1} & \sum x_i^{n+2} & \sum x_i^{n+3} & \cdots & \sum x_i^{2n} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ \sum x_i^3 y_i \\ \vdots \\ \sum x_i^n y_i \end{Bmatrix}$$

$$[XX]\{A\} = \{XY\}$$

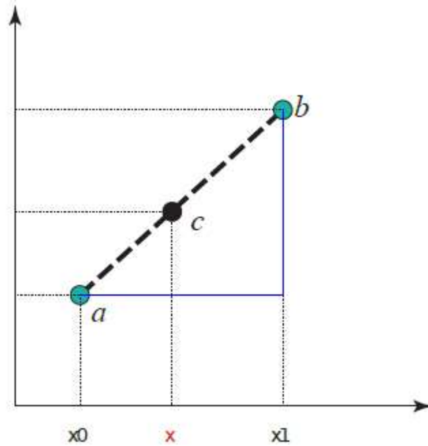
Interpolation

- Fitting a function through a set of data points

Newton Interpolation

Linear

- Uses similar triangles



$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1(x) - f(x_0)}{x - x_0}$$

- The linear-interpolation formula

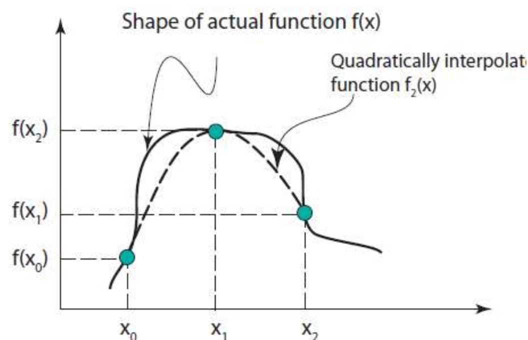
$$f_1(x) = f(x_0)\left(\frac{x_1 - x}{x_1 - x_0}\right) + f(x_1)\left(\frac{x - x_0}{x_1 - x_0}\right)$$

- The smaller the interval between x0 and x1, the better the approximation in general
- The subscript 1 in f1 denotes a first order interpolating polynomial

Quadratic

- Required to have same values as actual functions at data points

$$f_2(x) = a_0 + a_1 x + a_2 x^2$$



- Can write the equation as

$$f_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1)$$

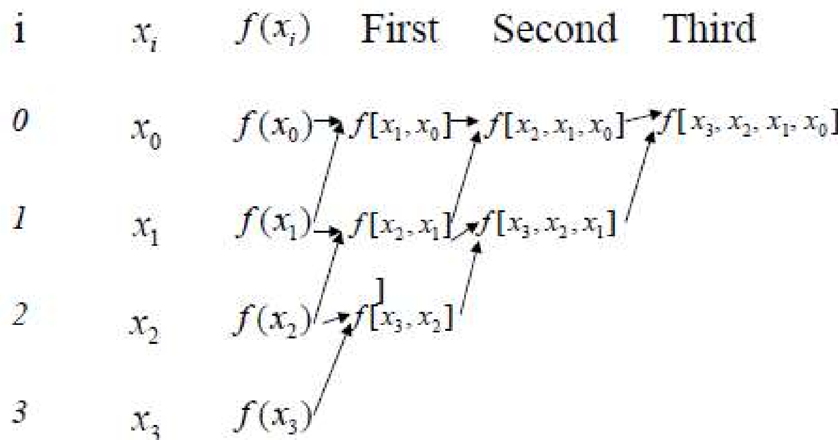
$$a_0 = b_0 - b_1 x_0 + b_2 x_0 x_1$$
$$a_1 = b_1 - b_2 x_0 - b_2 x_1$$
$$a_2 = b_2$$

- To do a quadratic interpolation, find all the b's in the equation by subbing in the function values at x0,x1,x2. By doing this you can find the values for b and so on with the following formulas, where the first one is called the first divided difference, and the second one is called the second divided difference and so on

$$b_1 = f[x_1, x_0] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$b_2 = f[x_2, x_1, x_0] = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0}$$

$$f[x_n, \ldots, x_0] = \frac{f[x_n, x_{n-1}, \ldots, x_1] - f[x_{n-1}, x_{n-2}, \ldots, x_0]}{x_n - x_0}$$



Lagrange Interpolating Polynomials

- Assume two functions L0(x) and L1(x) where

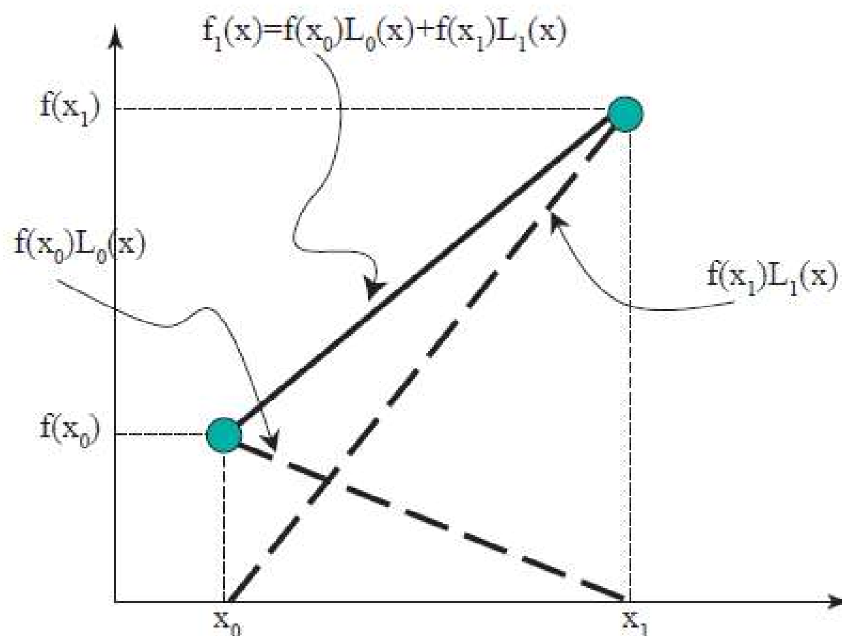
$$L_0(x) = \begin{cases} 1 & \text{if} \quad x = x_0 \\ 0 & \text{if} \quad x = x_1 \end{cases}$$

$$L_1(x) = \begin{cases} 0 & \text{if} \quad x = x_0 \\ 1 & \text{if} \quad x = x_1 \end{cases}$$

$$L_0(x) = \frac{(x - x_1)}{(x_0 - x_1)}, \qquad L_1(x) = \frac{(x - x_0)}{(x_1 - x_0)}$$

- The first order Lagrange polynomial is obtained by a linear combination of L0(x) and L1(x)

$$f_1(x) = f(x_0)L_0(x) + f(x_1)L_1(x)$$



- For a second order interpolation (three points), can do the same thing with three functions L0,L1,L2 where the function L0 = 1 if x = x0 and L0 = 0 if x = x1,x2 and similarly for the other two functions

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}$$

- Again, the Lagrange polynomial is a linear combination of L0, L1 and L2

$$f_2(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + f(x_2)L_2(x)$$

- In general, the nth order Lagrange polynomial (that can fit through n+1 data points) can be represented concisely as

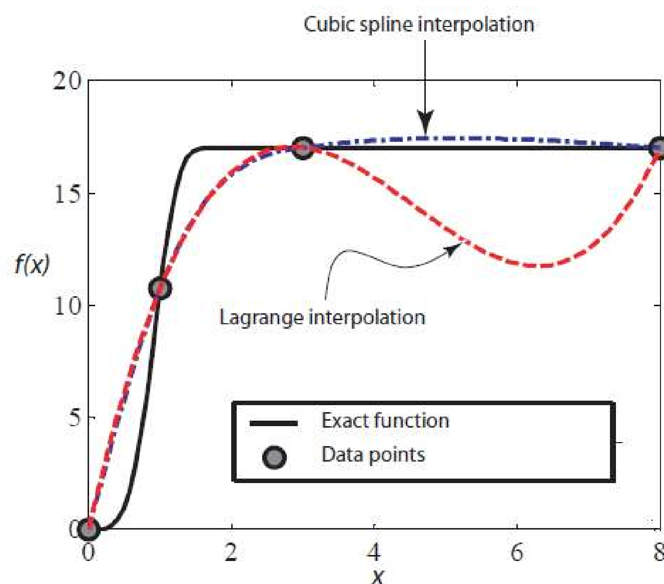
$$f_n(x) = \sum_{i=0}^{n} L_i(x) f(x_i)$$

$$L_i(x) = \prod_{j=0, j \neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

- Lagrange interpolating polynomial is just a different form of Newton interpolating polynomials, but is easier to program and slightly slower to compute

Spline Interpolation

- Newton and Lagrange can lead to erroneous results when there is an abrupt change in data



- Splines are made up of piecewise polynomials connecting only two data points
- $$S(x) = S_i(x) \text{ for } x_i < x < x_{i+1}.$$

Linear Spline

- Need to find the ai and bi for the equation Si(x) = ai + bi(x-xi)
- If there are n+1 data points, there are n intervals and hence 2n number of unknowns
- Constraints
  - Require Si(x) to have the values of f(xi) at x = xi
  - $$S_i(x_i) = a_i = f(x_i)$$
  - Require function values at adjacent polynomials to be equal at the interior knots
  - $$S_i(x_{i+1}) = S_{i+1}(x_{i+1})$$
    $$a_i + b_i(x_{i+1} - x_i) = a_{i+1}$$
    $$b_i = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

Quadratic Spline

- $$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2$$
- The function is continuous, but the first derivative is also continuous at x = xi
- Constraints
  - Require Si(x) to have the values of f(xi) at x=xi

    $$S_i(x_i) = a_i = f(x_i)$$
  - Function values at adjacent polynomials must be equal at interior knots

    $$S_i(x_{i+1}) = S_{i+1}(x_{i+1})$$
    $$a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = a_{i+1}$$

    For the sake of conciseness, let $h_i = x_{i+1} - x_i$, so

    $$a_i + b_i h_i + c_i h_i^2 = a_{i+1}$$
  - Derivative of Si(x) to be continuous at the interior nodes

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$$

$$b_i + 2c_i h_i = b_{i+1}$$

$$b_i = \frac{a_{i+1} - a_i}{h_i} - c_i h_i$$

- Therefore to find the quadratic spline
  - Set ai = f(xi)
  - Assume c0 = 0 (natural spline), this is the same as saying that the second derivative at x0 is zero.
  - Use this equation to find the values of the ci's

    $$c_j = \frac{1}{h_j}\left(\frac{a_{j+1}}{h_j} - a_j\left[\frac{1}{h_{j-1}} + \frac{1}{h_j}\right] + \frac{a_{j-1}}{h_{j-1}} - c_{j-1}h_{j-1}\right)$$

  - Use the equation for bi to obtain all the bi's

Cubic Spline

- Now Si(x) and its first, second derivatives are all continuous at x = xi

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$$

- Constraints
  - Si(xi) = ai = f(xi)
  - Function values are continuous at internal nodes (i=0 … n-2)

    $$S_i(x_{i+1}) = S_{i+1}(x_{i+1})$$

    $$a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 = a_{i+1}$$

  - Derivative is also continuous at interior nodes for (i=0 … n-2)

    $$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$$

    $$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1}$$

  - Second derivative also continuous at interior nodes for (i=0 … n-2)

$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$$

$$c_i + 3d_ih_i = c_{i+1}$$

- Now we have 4n-2 conditions but still need 2 more conditions to get 4n unknowns
  - Clamped Spline – If you know more information about the function you are trying to approximate you can generate the two constraints for the double derivative at end points
  - Natural Spline – No information, just assume that the double derivative is zero at two end points, c0 = 0, cn = 0

- With all 4n conditions, can rearrange for
  - From the second derivative equation

$$\boxed{d_i = \frac{c_{i+1} - c_i}{3h_i}}$$

  - From the above and the function value continuity equation

$$\boxed{b_i = \frac{1}{h_i}(a_{i+1} - a_i) - \frac{h_i}{3}(2c_i + c_{i+1})}$$

  - Sub into first derivative continuity

$$b_{i+1} = b_i + (c_i + c_{i+1})h_i$$

  - To get an equation purely in terms of ai's and ci's

$$h_{j-1}c_{j-1} + 2(h_j + h_{j-1})c_j + h_jc_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) + \frac{3}{h_{j-1}}(a_{j-1} - a_j)$$

  - Now using our clamped/natural condition (assume natural for this ex ample, can solve a system of linear equations (tridiagonal)

$$[A]=\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & 0 & \cdots & 0 \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

$$\{X\} = \begin{Bmatrix} c_0 \\ c_1 \\ \vdots \\ \vdots \\ \vdots \\ c_n \end{Bmatrix}$$

$$\{C\} = \begin{Bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) + \frac{3}{h_0}(a_0 - a_1) \\ \frac{3}{h_2}(a_3 - a_2) + \frac{3}{h_1}(a_1 - a_2) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) + \frac{3}{h_{n-2}}(a_{n-2} - a_{n-1}) \\ 0 \end{Bmatrix}$$

- Therefore to solve for cubic spline
  - Set all the ai = f(xi)
  - Solve the tridiagonal linear system to get the ci’s
  - Substitute this in to find bi
  - Substitute in to find di

Regression

- An alternative way of solving the least squares problem

$$\begin{bmatrix} \sum_{i=1}^{N} x_i^2 & \sum_{i=1}^{N} x_i \\ \sum_{i=1}^{N} x_i & N \end{bmatrix} \begin{Bmatrix} a \\ b \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^{N} x_i y_i \\ \sum_{i=1}^{N} y_i \end{Bmatrix}$$

- Can be shown that

$$b = \overline{y} - a\overline{x}, \quad a = \frac{\sum_{i=1}^{N}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

- Linear least squares regression is a method of curve fitting, can also be extended to multiple regression, where $\hat{y}$ is a function of n independent values.

Multiple Linear Regression

- When fitting a model of n variables, define xij the ith observation of the jth variable

$$\hat{y}_i = a_0 + a_1 x_{i1} + a_2 x_{i2} + \cdots + a_j x_{ij} + \cdots + a_n x_{in}$$

$$\begin{bmatrix} N & \sum x_{i1} & \sum x_{i2} & \dots & \sum x_{in} \\ \sum x_{i1} & \sum x_{i1}x_{i1} & \sum x_{i2}x_{i1} & \dots & \sum x_{in}x_{i1} \\ \sum x_{i2} & \sum x_{i1}x_{i2} & \sum x_{i2}x_{i2} & \dots & \sum x_{in}x_{i2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{in} & \sum x_{i1}x_{in} & \sum x_{i2}x_{in} & \dots & \sum x_{in}x_{in} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum y_i \\ \sum x_{i1}y_i \\ \sum x_{i2}y_i \\ \vdots \\ \sum x_{in}y_i \end{Bmatrix}$$

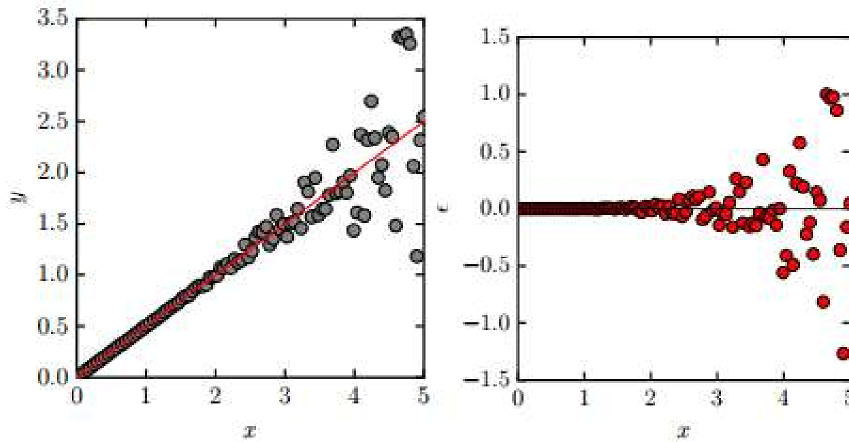
Assumptions

- Y is a linear function of all zi
  - This is obvious as we build our model as a linear function y = az1 + bz2 + c
  - Z1 and z2 can be non-linear, but y must be linear in zi

$$z_1 = x_1^3, \quad z_2 = \sin(x_2)$$

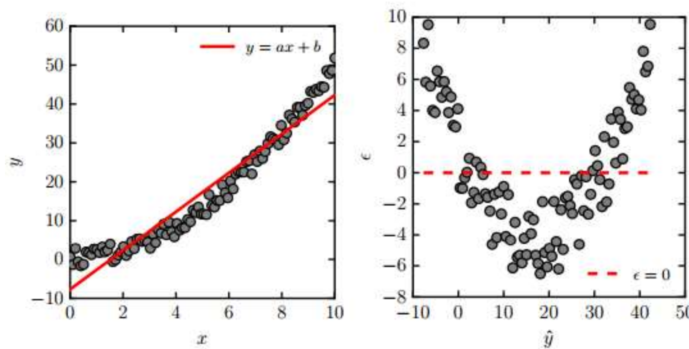
- No autocorrelation
  - This means consecutive errors in the model are not correlated
  - Each predictor variable is independent of each other
- Homoscedasticity
  - This means constant variation in error

$$\mathrm{Var}(\epsilon_i) = \sigma_i^2 = \sigma^2, \forall i$$



- Normally Distributed Errors
  - We are minimizing the sum of squares, if one 'square' is very large, then we may not make sensible predictions with our model
- Breaking assumptions
  - Linearity
    - Detect by plotting y_i vs $\hat{y}_i$, points should roughly be symmetric about a diagonal line
    - And/or ϵ_i vs $\hat{y}_i$, points should roughly be symmetric about horizontal line
    - For

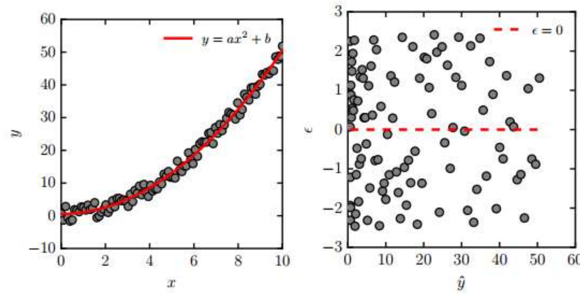
$$\hat{y} = ax + b, \quad y = \hat{y} + \epsilon$$



- It can be seen that the points are not symmetric about the given lines, this can be fixed by a variable transform

$$\hat{y} = az + b, \quad z = x^2$$

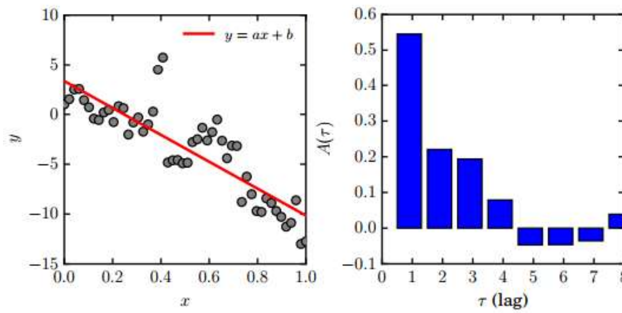
-



- Autocorrelation
  - There should be no autocorrelation of error

$$\mathrm{A}(\tau) = \sum_{i=1}^{N} \frac{(\epsilon_i - \overline{\epsilon})(\epsilon_{i+\tau} - \overline{\epsilon})}{\sigma_\epsilon} \approx 0, \quad \forall \tau \in \mathbb{N}$$

  - This can be detected by plotting $A(\tau)$ vs τ

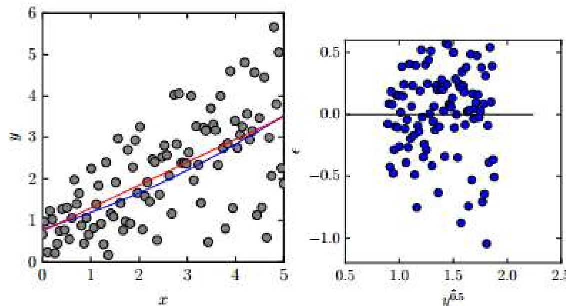


  - This can be fixed by adding a lag if $0.2 < A(\tau = j) < 0.5$

$$\hat{y}_i = ax + b + cx_{i-j}$$

  - Where xi-j is the lag, that is a delay of x by j observations. This reduces degrees of freedom by j, which may affect R^2
  - For example might see a lag for every 4 or 12 for quarterly/monthly data
- Homoscedasticity
  - The variance of the model error epsilon should be constant for all predictor variable values.
  - Detect by plotting ϵ_i vs $\hat{y}$
  - Fix by taking the log or square root of the observed variable, for example

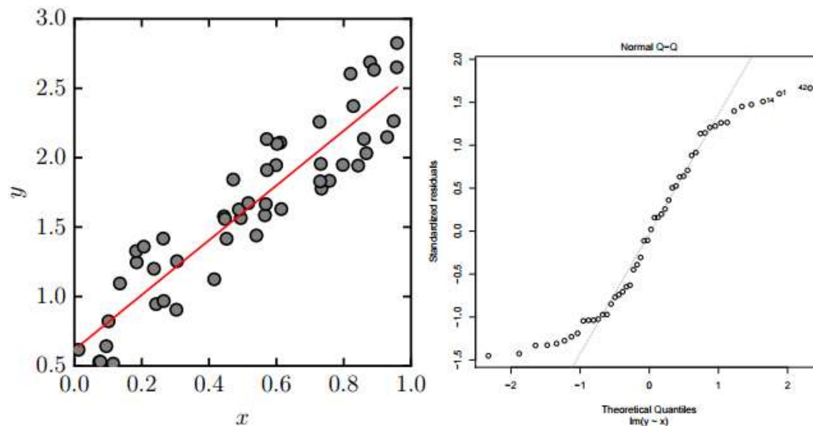
$$\sqrt{y} = y^* = ax_i + b$$



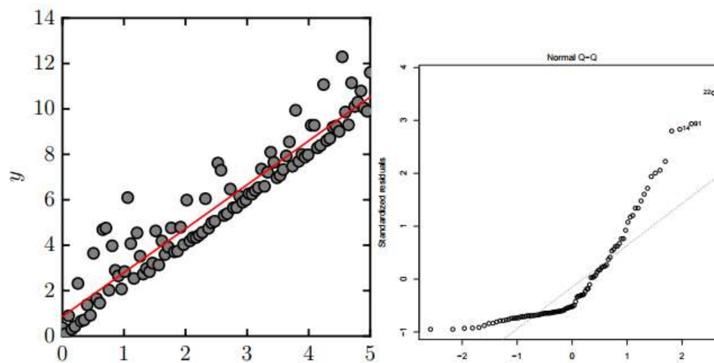
- Normality
  - The model errors ϵ_i must come from a normal distribution, i.e the probability of observing an error $\epsilon_i = \zeta$ is

$$P(\epsilon_i = \zeta) = \frac{1}{\sqrt{2\pi\sigma_\epsilon^2}} \exp\left(- \frac{\zeta^2}{2\sigma_\epsilon^2} \right)$$

  - Detect by plotting the quantiles of standardised residuals against theoretical residuals i.e calculate the quantiles of $\frac{\epsilon_i}{\sigma_\epsilon}$ and plot against the quantiles of N(0,1)
  - This is a Q-Q plot



  - Example of a bad violation of normality, where outliers are affecting the fit



  - Normality is often violated because linearity is violated. Therefore transformations of variables might help
  - Another possibility is to get more data, which appeals to the central limit theorem (when you have enough independent data the mean approaches a normal distribution

Goodness of Fit

- The 'goodness' of the model is contained in the R^2 value, the aim is to minimise our 'cost function S

$$S = \sum_{i=1}^{N} \epsilon_i^2$$

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \overbrace{ax_i - b}^{\hat{y}_i})^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}$$

$$= 1 - \frac{\text{SSE}}{\text{SST}}$$

- The numerator is the residual sum of squares
- The denominator is the total sum of squares
- Therefore R^2 is the ratio of variance explained by the model over the total variation in observed values, known as the coefficient of determination, also called the squared Pearson correlation coefficient.

$$R^2 = r_{y,\hat{y}}^2$$

$$r_{y,\hat{y}}^2 = \frac{\text{Cov}(y,\hat{y})}{\text{Var}(y)\text{Var}(\hat{y})}$$

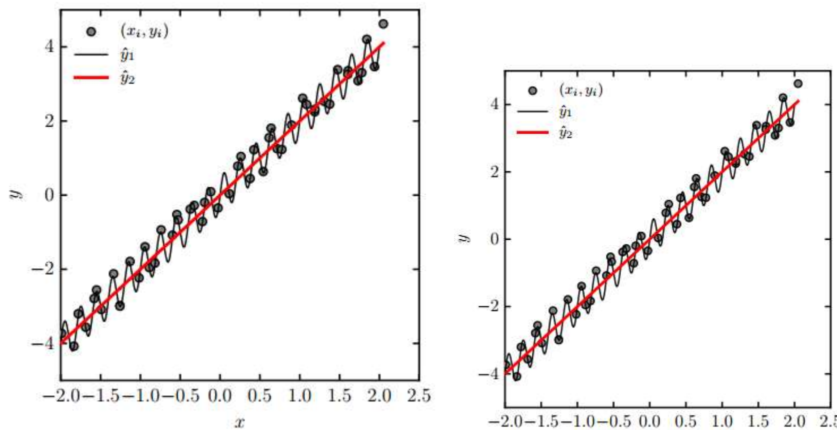
- If R^2 = 1, all variation in yi is accounted for by the model: SSE = 0
- If R^2 = 0, no variation in yi is accounted for by the model: SSE = SST
- If R^2 < 0, the model is worse than using the sample mean as our model $\hat{y} = \bar{y}$
- R^2 can be used for other regression types, where $\hat{y}$ is a model we choose: polynomial/multiple/a combination of the two

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}$$

- R^2 is sensitive to the number of variables n, can wrongly increase if you increase the number of variables
  - As a rule of thumb, 20 data points per predictor variable are required
  - If N = n+1, we have 1 coefficient for each data point, which is overfitting the data
  - Can account for this by using the adjusted R^2, which penalises models that use more variables. As n approaches N-1, R^2 bar approaches negative infinity and as N/n approaches infinity, we recover the original coefficient

$$\overline{R}^2 = R^2 - (1 - R^2)\frac{n}{N - n - 1}$$

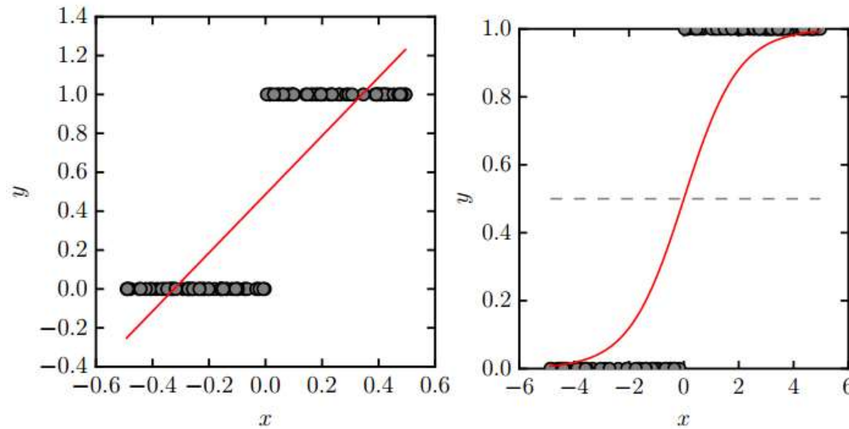
- High R^2 isnt always better, if statistical significance of coefficients can be shown then you can still say the model explains X amount of variance, some subjects such as psychology and medicine naturally have a more random variation



-

Regression for classification

- Can use predictor variables to classify things like images, observe things like failure or not failure
- Common theme
  - Observed variable is discrete and can be assigned labels
  - $y_i = (0,1,\dots,g-1)$ where g is the number of groups
  - Predictor variables xi are continuous (wings, beaks, etc)
  - A continuous model would not make sense as invalid values of yi could occur
  - $$\hat{y} = b + a_1 x_1 + \cdots + a_n x_n = b + \mathbf{a} \cdot \mathbf{x}$$
  - Instead the probability of whether yi is a member of a given group is regressed
- Example: linear line doesn't tell much about the data, but a sharp cutoff is better



- Can convert from a model with xi in a domain of (-infinity, infinity) to [0,1] by the logistic function for a binary case of y in {0,1}

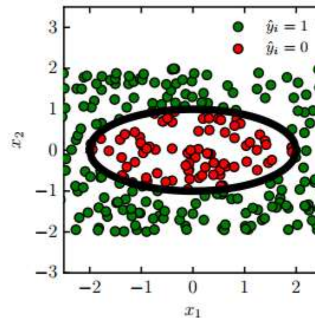
$$h_{\mathbf{a},b}(x_1, \ldots, x_n) = \frac{1}{1 + \mathrm{e}^{-\mathbf{a}\cdot\mathbf{x}-b}}$$

- Where ha,b(x) takes the predictor variables and returns a probability of belonging to group 1
- For ha,b(x) = 1, it implies the model is certain that yi = 1 (image is penguin for example)
- For ha,b(x) = 0, it implies the model is certain that yi = 0 (image not penguin)
- For a value of 0.5, implies the model cannot decide
- Therefore, the Logistic Regression model is

$$\hat{y}_i = \begin{cases} 1, & \text{if } h_{a,\mathbf{x}} > 0.5 \\ 0, & \text{if } h_{a,\mathbf{x}} \leq 0.5 \end{cases}$$

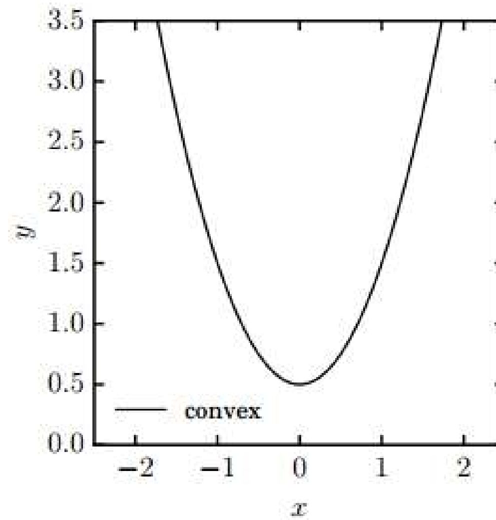
- Example

  Suppose our model is $y^* = a_1 x_1^2 + a_2 x_2^2 + b$, with
  $a_1 = 1,\ a_2 = 4,\ b = -4$,
  then we predict $\hat{y}_i = 1$ if: $x_1^2 + 4x_2^2 > 4$

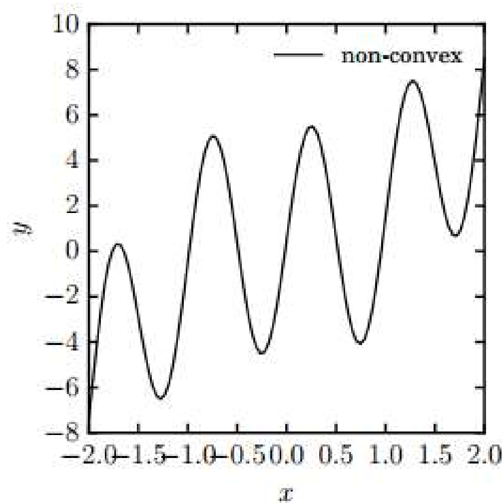


- The Linear Regression cost function is no longer possible as the logistic function is now included

  - Not quadratic in parameters (cannot form matrix as before by setting derivatives to 0)
  - Non-convex: highly non-linear so there are many minima
- For a convex function (mid point value in every interval does not exceed arithmetic mean) we can find local minima



- Non convex function makes it hard to find the local minima



- Convex cost function
  - define:

$$S^*(\mathbf{a}, b, \mathbf{x}, y_i) = \begin{cases} -\log(h_{\mathbf{a},b}(x_1, \ldots, x_n)), & \text{if } y_i = 1 \\ -\log(1 - h_{\mathbf{a},b}(x_1, \ldots, x_n)), & \text{if } y_i = 0. \end{cases}$$

  - **convex function**:

$$S(\mathbf{a}, b) = \sum_{i=1}^{N} S^*(\mathbf{a}, b, y_i)$$

- The worse the model parameters **a** and b, the higher the cost S (We want to minimize this for parameters **a** and b
- We can see from the convex cost function that if yi=0 and the prediction $\hat{y}_i = 1$ the model is strongly penalised, the same thing happens in the opposite scenario
- Need an algorithm to minimize S by tuning model parameter **a**, b
  - Define alpha, a selectable 'learning rate'
  - Randomly initialise **a**, b = {N(0,1), … , N(0,1)}T
  - While not converged (parameters don't change to a given tolerance)
    - Calculate cost function S(**a**,b)
    - Calculate cost function gradient deltaS(**a**,b)
    - Update params: {**a**,b} = {**a**,b} – alpha*deltaS(**a**,b)
  - Output model parameters: {**a,**b}
- Gradient descent works because –deltaS(**a**,b) is a vector that points in the direction of the fastest negative change
- Summary
  - With N observations for a variable y = {0,1}, we want a model that tells us which group future observations will be in
  - Given continuous predictor variables x1, …, xn, we can define intercept b and n slope parameters a1, …, an
  - Use the logistic function ha,b(x) to build a model for y prediction
  - Define a convex cost function S(**a**,b)
  - Tune model parameters **a**,b

Integration

Trapezoidal Rule

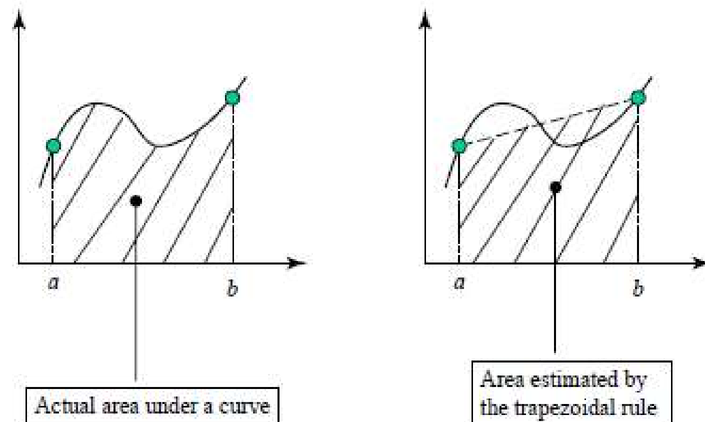
- Best you can do with two points is to fit a straight line through and find area underneath it
- Use trapezoidal rule to estimate area underneath a line, can be derived by integrating a first order Lagrange polynomial from x0 to x1

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{\Delta}{2}\left(f(x_i) + f(x_{i+1})\right)$$

- Error of approximation is: (where zeta is some number between xi and xi+1

$$\frac{\Delta^3}{12} f^{(2)}(\xi)$$

- If the size of the interval delta is large, the error is large as well. Can decrease error by splitting into smaller subdomains (usually of equal length)



- With many points, can do trapezoidal rule between all the points with distance of equal size delta

$$\int_a^b f(x)dx \approx \frac{\Delta}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b)\right)$$

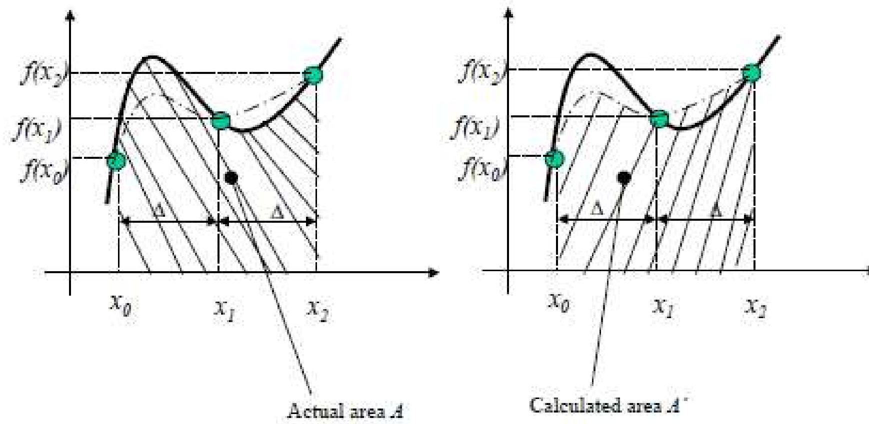
Simpson's Rule

- Trapezoidal rule is derived by a first order Lagrange approximation through two points
- Simpson's Rule uses a second order Lagrange fit through three points
- Formula for second order Lagrange polynomial, the last term is the error term

$$\begin{aligned} f_2(x) \quad = \quad & \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f(x_0) \\ & + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f(x_1) \\ & + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f(x_2) \\ & + \frac{1}{6}(x-x_0)(x-x_1)(x-x_2)f^{(3)}(\xi(x))dx \end{aligned}$$

- Integrating between x0 and x2, we can get

$$\int_{x_i}^{x_{i+2}} f_2(x) = \frac{\Delta}{3}\left(f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right)$$



- General formula for an arbitrary amount of data points (only works if the total number of data points (N+1) is odd and the number of intervals (N) is even

$$\int_{x_0}^{x_N} f(x)dx \approx \frac{\Delta}{3}\left(f(x_0) + 4\sum_{i=1 i odd}^{N-1} f(x_i) + 2\sum_{i=1 i even}^{N-2} f(x_i) + f(x_N)\right)$$

Differentiation

First Order

Forward Difference Approximation

- To find a formula to differentiate, can use Taylor series approximation

$$\begin{aligned} f(x_{i+1}) = f(x_i) \quad &+ \quad f'(x_i)(x_{i+1} - x_i) + \frac{1}{2!}f''(x_i)(x_{i+1} - x_i)^2 \\ &+ \quad \frac{1}{3!}f'''(x_i)(x_{i+1} - x_i)^3 + \ldots , \end{aligned} \tag{9}$$

where

$$x_{i+1} = x_i + \Delta$$

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

- Rearranging for the first derivative, we can obtain

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)} - \frac{1}{2!}f''(x_i)(x_{i+1} - x_i) + \frac{1}{3!}f'''(x_i)(x_{i+1} - x_i)^2 + \ldots \qquad (10)$$

- Therefore it can be approximated via a FDA (Forward Difference Approximation) by ignoring the second+ derivative terms (these become the error)

$$\boxed{f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)}}$$

- The leading term in truncation error is

$$E_{FDA} = \frac{1}{2!}f''(x_i)(x_{i+1} - x_i)$$

Backward Difference Approximation

- Similarly, can use Taylor series to obtain an expression for f(xi-1) and rearrange to get the BDA (Backward Difference Approximation)

$$\boxed{f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{(x_i - x_{i-1})}}$$

$$E_{BDA} = \frac{1}{2!}f''(x_i)(x_i - x_{i-1})$$

Central Difference Approximation

- Subtracting the equivalent equation for BDA from equation (10), we can obtain the CDA (Central Difference Approximation)

$$\boxed{f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}}}$$

$$E_{CDA} = f''(x_i)\frac{(x_{i+1} - x_i)^2 - (x_i - x_{i-1})^2}{2!(x_{i+1} - x_{i-1})}$$

- Assuming all the xi are equally spaced, the FDA, BDA, and CDA can be simplified to

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta}$$

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta}$$

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta}$$

Second Order

Central Difference Approximation

- Again rearranging for the second order derivative using Taylor series expansion

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{(x_{i+1} - x_{i-1})^2}$$

- Leading order error is O(delta^2)

Wave Number Analysis

- Assume equidistantly spaced grid with length Lx and Nx number of points
- Spacing of Delta = Lx/Nx
- Values of fi = f(xi) are given on each grid point xi = delta*i, i = 0,1,2, … Nx
- Central Difference Approximation becomes (where O(delta^n) is the truncation error of scheme)

$$\sum_{j}^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{h} \left(\sum_{j}^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n) \ , i = 1, \dots, N_x$$

- Coefficient alphaj and aj are typically derived using Taylor-series expansion and chosen to given the largest possible exponent n = 2(Nalpha + Na). This minimizes the truncation error.
- For Nalpha = 0, the derivative is discretised with standard, or explicit finite-different stencil
  - f'i depends only on function values at neighbouring nodes
  - Small Na requires less operations than wider stencils
  - Accuracy lower as n = 2Na

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{h}\left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j})\right) + \mathcal{O}(\Delta^n) \ , i = 1, \ldots, N_x$$

- For Nalpha =/= 0, the derivative is discretised with compact, or Pade finite-difference stencil
  - f'i depends on function values AND derivatives at neighbouring nodes
  - Requires more operations (to find the solution of a banded matrix in solving the system of equations)
  - Accuracy is higher because n = 2(Nalpha + Na)
  - Example: five point stencil with Nalpha = 1 and Na = 2 results in a 6th-order accurate method instead of 4th-order method
- A finite difference stencil is not judged by only order-of-accuracy: dissipation and dispersion errors can be significant
- Assuming an equidistant grid and a periodic f(x) on interval [0,Lx], Fourier representation is

$$f(x) = \sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x}{\Delta}\right) \ ,$$

where k_m is the wavenumber, defined as

$$k_m = 2\pi m\Delta/L_x \ , \quad m = 0, 1, 2, ..., N_x/2$$

spanning interval $[0, \pi]$.

- Can take the derivative of this by taking the derivative of each Fourier coefficient individually (due to linearity).

$$f'(x) = \frac{d}{dx}\left[\sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x}{\Delta}\right)\right] = \sum_m \mathrm{i}\frac{k_m}{\Delta}\hat{f}_m \exp\left(\mathrm{i}k_m \frac{x}{\Delta}\right) \ , \quad (20)$$

$$f'(x) = \sum_m \hat{f}'_m \exp\left(\mathrm{i}k_m \frac{x}{\Delta}\right) \ ,$$

- Error made by finite-difference scheme can be quantified by comparing the approximation $(\hat{f}'_m)_{FD}$ with the exact fourier mode $\hat{f}'_m$
- Example

Thus, the modified wavenumber for this FDA scheme is

$$k_m^{mod} = \mathrm{i}\left[1 - \exp\left(\mathrm{i}k_m\right)\right]$$

Using $\exp(\mathrm{i}x) = \cos(x) + \mathrm{i}\sin(x)$

$$\begin{aligned} k_m^{mod} &= \mathrm{i}\left[1 - \left(\cos\left(k_m\right) + \mathrm{i}\sin\left(k_m\right)\right)\right] \\ &= \sin\left(k_m\right) + \mathrm{i}\left[1 - \cos\left(k_m\right)\right] \end{aligned}$$

Real and imaginary parts of modified wavenumber:

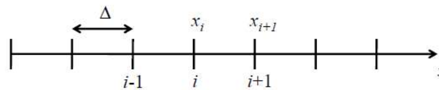
$$k_{m_R}^{mod} = \sin\left(k_m\right) \quad \Rightarrow \quad \text{Phase/Dispersion error}$$
$$k_{m_I}^{mod} = 1 - \cos\left(k_m\right) \quad \Rightarrow \quad \text{Amplitude/Dissipation error}$$

Illustrate with simple example: Consider the FDA

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta}$$

Represent the function with Fourier series

$$f(x_i) = \sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$



$$f(x_{i+1}) = \sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right)$$

Insert Fourier representation of $f(x_{i+1})$ and $f(x_i)$ into FDA:

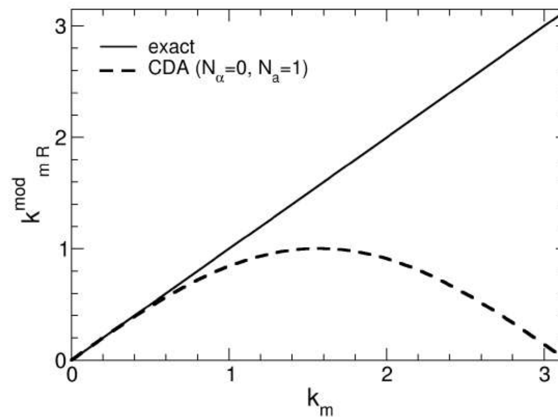
$$\begin{aligned} f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta} &= \frac{1}{\Delta}\sum_m \hat{f}_m \left[\exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right) - \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)\right] \\ &= \sum_m \frac{1}{\Delta}\left[\exp\left(\mathrm{i}k_m \frac{\Delta}{\Delta}\right) - 1\right] \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \end{aligned}$$

The exact derivative is

$$f'(x_i) = \sum_m \frac{\mathrm{i}k_m}{\Delta} \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$

Compare:

$$\frac{\mathrm{i}k_m}{\Delta} = \frac{1}{\Delta}\left[\exp\left(\mathrm{i}k_m\right) - 1\right] \equiv \frac{\mathrm{i}k_m^{mod}}{\Delta}$$

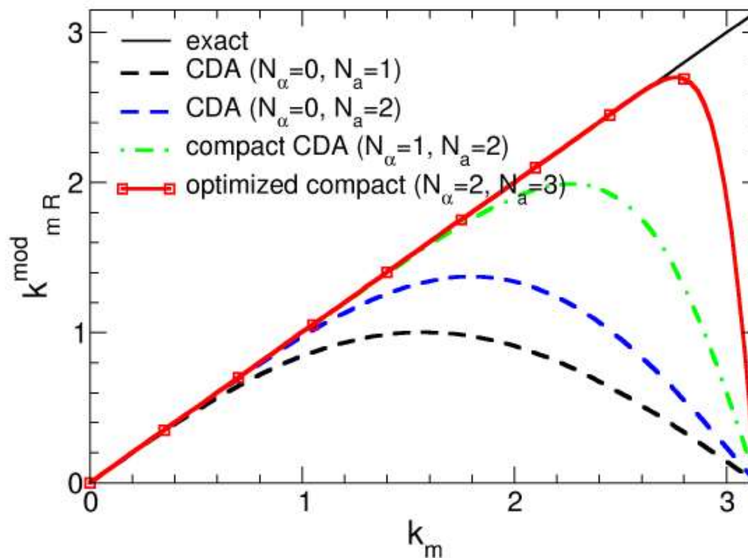


- The modified wavenumber for most general finite-difference scheme

$$\sum_{j}^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{h} \left(\sum_{j}^{N_a} a_j (f_{i+j} - f_{i-j}) \right) :$$

$$\boxed{k_m^{mod} = \frac{\sum_j^{N_a} 2a_j \sin(jk_m)}{1 + \sum_j^{N_\alpha} 2\alpha_j \cos(jk_m)}}$$

- Plots of the real part



Ordinary Differential Equations

- Differential equation

$$\frac{dx}{dt} = f(t, x)$$

- Domain

$$a \leq t \leq b$$

- Initial condition

$$x(t = a) = \alpha$$

Euler Method

Forward Euler Method

- Considering Taylor's theorem

$$x(t_{n+1}) = x(t_n) + hf(t_n, x_n) + \frac{h^2}{2}\frac{d^2x}{dt^2}(t_n)$$

$$t_{n+1} - t_n = h$$

- If we assume h is small, we can neglect the second order term and get the formula for Euler's method where xn is a numerical approximation of exact solution x(tn)

$$\boxed{x_{n+1} = x_n + hf(t_n, x_n)}$$

- Called explicit/forward Euler, f only contains tn and xn
- Can often be unstable

Backward Euler Method

- More stable, backward/implicit Euler scheme
- $$\boxed{x_{n+1} = x_n + hf(t_{n+1}, x_{n+1})}$$
- Implicit because xn+1 exists on both sides of the equation.
- If f is a simple function, it might be possible to obtain an explicit expression for xn+1
- If f is a complicated function, use root finding methods like Newton-Rahpson to solve for xn+1

Crank Nicolson method

- Can also obtain the solution to dx/dt = f(t,x) by integration

- $$x(t) = x(t_n) + \int_{t_n}^{t_{n+1}} f\left[t, x(t)\right] dt$$
- Using the trapezoidal rule for integration

$$\int_{t_n}^{t_{n+1}} f(t, x(t)) dt = \frac{h}{2}\left[f(t_n, x_n) + f(t_{n+1}, x_{n+1})\right] + O(h^3)$$

$$\boxed{x_{n+1} = x_n + \frac{h}{2}\left[f(t_n, x_n) + f(t_{n+1}, x_{n+1})\right]}$$

- Again, need to solve for xn+1 using root finding methods if it is complicated

Runge-Kutta Methods

- Can be written as
- $$x_{n+1} = x_n + \phi(x_n, t_n, h)h$$
- Where phi is known as incremental function and can be interpreted as the slope used to predict the new value of x

$$\phi = a_1k_1 + a_2k_2 + a_3k_3 + a_4k_4 + \cdots a_Nk_N$$

$$\begin{aligned}
k_1 &= f(t_n, x_n) \\
k_2 &= f(t_n + p_1h, x_n + q_{11}k_1h) \\
k_3 &= f(t_n + p_2h, x_n + q_{21}k_1h + q_{22}k_2h) \\
k_4 &= f(t_n + p_3h, x_n + q_{31}k_1h + q_{32}k_2h + q_{33}k_3h) \\
\vdots \;\; &\vdots \;\; \vdots \\
k_N &= f(t_n + p_{N-1}h, x_i + q_{N-1,1}k_1h + q_{N-1,2}k_2h + \cdots + q_{N-1,N-1}k_{N-1}h)
\end{aligned}$$

- For N = 1, we get the first order Runge-Kutta scheme, which is just the same as the Euler integration scheme

- The 4th order Runge Kutta scheme is by far most popular numerical method for solving ODEs.

  The formula for this scheme can be written as

  $$x_{n+1} = x_n + \left(\frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4\right)h$$

  where

  $$\begin{aligned} k_1 &= f(t_n, x_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2\right) \\ k_4 &= f(t_n + h, x_n + hk_3) \end{aligned}$$

Stability and Error Analysis

- Consider the below, for most engineering problems, the real part of lambda is negative so the solution will decay over time

$$\frac{dx}{dt} = \lambda x$$

- Applying the Euler method with timestep deltat = h

$$x_{n+1} = x_n + \lambda h x_n$$

or $$x_{n+1} = (1 + \lambda h)\, x_n$$

- Can write error at any time step n

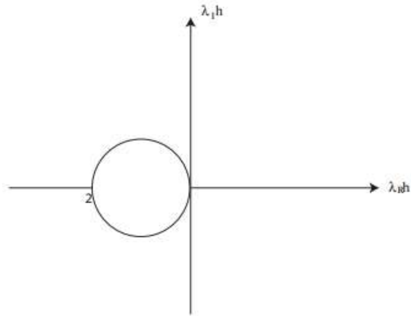
$$x_n = x_0 (1 + \lambda h)^n$$

$$x_n = x_0 (1 + h\lambda_R + ih\lambda_I) = x_0 \sigma^n$$

- Where sigma is the amplification factor, and the requirement for the stability of the scheme is for the magnitude of sigma to be less than 1

$$\begin{aligned} |\sigma| &\leq 1 \\ |\sigma|^2 &= (1 + h\lambda_R)^2 + (h\lambda_I)^2 \leq 1 \end{aligned}$$

- Stability diagram is shown below, where if lambda is real and negative, for the numerical method to be stable then h <= 2/|lambda|



- Different stability regions for different orders of R-K

For the second-order R-K method, we get

$$1 + \lambda h + \frac{\lambda^2 h^2}{2} - e^{i\theta} = 0$$

4th order R–K method, stability region obtained from

$$\lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24} + 1 - e^{i\theta} = 0$$