Numerical Methods in Engineering

1 - Numerical Algorithms

Computer Representation of real numbers

Round of Errors

Rounding

Chopping

Errors in numerical methods

Properties of Algorithms

Convergence

Efficiency

Conditioning and stability

2 - Solving Nonlinear equations

Finding Zeros of a nonlinear function

- There is a lack of closed formulas
- -> we develop iterative methods
- We do not reach the exact solution but we look for an approximation
- We often require a good first guess
- Roots of a method are points where y = 0

Bisection Method

- To cut in half
- Bracketing method - we choose an interval that changes sign
    - A root (zero) must be in there if function is continuous
- Not really usable with multiple variables
- Systematically reduce the width of the bracket
- Bisection doesn't considerate function behaviour

- **Start**: interval $I^{(0)} = (a, b)$ such that $f(a)f(b) < 0$; compute the midpoint $x^{(0)} = \frac{a+b}{2}$.
- Start the **loop** $k = 1$:
    - evaluate $f(x^{k-1})$ and check the sign of $f(a)^{(k-1)}f(x)^{(k-1)}$, if
        - < 0, the root must be in $I^{(k)} = (a^{(k-1)}, x^{(k-1)})$, so we set $a^{(k)} = a^{(k-1)}$ and $b^{(k)} = x^{(k-1)}$;
        - > 0, the root must be in $I^{(k)} = (x^{(k-1)}, b^{(k-1)})$, so we set $a^{(k)} = x^{(k-1)}$ and $b^{(k)} = b^{(k-1)}$;
        - $= 0$, $(x)^{(k-1)}$ is the zero.
    - compute the midpoint $x^{(k)} = \frac{a^{(k)}+b^{(k)}}{2}$.
    - **Stop** check: is size of the interval smaller than a given value, e.g., $|I^{(k)}| < \epsilon$?
        - if yes, $x^{(k)}$ is the final approximation of α, and **exit the loop**
        - if no, **iterate**: $k = k + 1$.
- **Output**: $x^{(k)}$

```
function [x,it] = Bisection(fun, x0, x1, eps)
    % Input:
    %      fun (function handle): the function to find the root for
    %      x0  (float):           the lower bound of the interval
    %      x1  (float):           the upper bound of the interval
    %      eps (float):           the given tolerance
    Nmax = 5000;

    % Initialising the iteration number and setting
    % an error bigger than eps to enter in the loop
    it  = 0;
    err = eps+1;

    % Initialising the considered interval bounds
    a = x0;
    b = x1;

    % Defining the iteration loop that is computing the bisection
    while ((err>eps)&&(it<Nmax))
        % Evaluate the midpoint of the interval
        mid=(a+b)*0.5;
        % Check which part of the interval to select
        if (fun(a)*fun(mid)<0)
             b=mid;
        else
            a=mid;
        end
        % Computing the error (the stopping criterion)
        % and incrementing the number of iterations
        err = 0.5*abs(b-a);
        it  = it+1;
    end

    % Informing the user of the function output
    if ((err>eps) || (it>Nmax))
        % Case where the function failed to find the zero before the
        % allowed number of iterations
        error(sprintf("After %d iterations, no root has been found with the desired accuracy.", it))
    else
        % Informing the user
        fprintf("The number of iterations required to achieve the tolerance %f is %d\n", eps, it);
        % Estimate the zero as the mid point of the found interval
        x = (a+b)*0.5;
        fprintf("The root is located within the interval [%d, %d]\n", a, b);
    end
end
```

The sequence necessarily converges since the interval is halved every iteration, so $|I^{(k)}| = (1/2)^k|I^{(0)}|$. Thus, the error at the step k satisfies

$$|e^{(k)}| = |x^{(k)} - \alpha| < \frac{1}{2}|I^{(k)}| = \left(\frac{1}{2}\right)^{k+1}(b-a).$$

Iteration required to have $|e^{(k)}| < \epsilon$: $k_{\min} = \log_2\left(\frac{b-a}{\epsilon}\right) - 1$

- This approach is an absolute criteria (bad) - relative would be to check residual |f(x^k)|<e^-12

Newton Method & variations

- With additional information we can get more efficient than bisection
- We take the first derivative of our initial guess f'(x^k) (e.g. the tangent)
- We calculate the tangents interception with the x-axis
- We use this intersection point as our new iterate x^(k+1)

If f is differentiable, tangent equation is $y(x) = f(x^{(k)}) + f'(x^{(k)})(x - x^{(k)})$.

The point where it crosses the x-axis is $x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$, provided that $f'(x^{(k)}) \neq 0$. e.g. **Newton-Raphson Method**

Rate of convergence Provided an iterative method converges, we could be interest in determining how fast it does. We introduce the concept of **speed (or rate) of convergence**. A method is said to be

- *linearly convergent* if there exist a constant $\rho < 1$ such that $|x^{(k+1)} - \alpha| \leq \rho|x^{(k)} - \alpha|$, for all k sufficiently large.
- *quadratically convergent* if there exist a constant M such that $|x^{(k+1)} - \alpha| \leq M|x^{(k)} - \alpha|^2$, for all k sufficiently large.
- *superlinearly convergent* if there is a sequence of constants $\rho_k \to 0$ such that $|x^{(k+1)} - \alpha| \leq \rho_k|x^{(k)} - \alpha|$, for all k sufficiently large.

High order convergence rate can be defined as $|x^{(k+1)} - \alpha| \leq C|x^{(k)} - \alpha|^p$, with $p \geq 2$. Given an order of convergence p, smaller the asymptotic error constant is, faster the convergence is. However, we do not need to know this constant, but knowing p is sufficient.

Convergence If $x^{(0)}$ is *sufficiently close* to α and α is a simple zero, the Newton method converges. If also f is differentiable up to second order, the Newton method converges quadratically. Suitable values for $x^{(0)}$ can be found by bisection, graphics. If f is linear, Newton method converges in 1 step. If $m > 1$, the convergence is only linear.

```
function [x, r, it, xxs] = NewtonMeth(fun, dfun, x0, itMax, eps)
    %% INPUT:
    % fun       function handle,
    % dfun      derivative function handle,
    % x0        inital value,
    % itMax     number of maximum iterations
    % eps       prescribed convergence tolerance Îµ
    %% OUTPUT:
    % x         approximate result,
    % r         residual at the last iteration,
    % it        number of perfomred iterations,
    % eps       vector containing all computed x^(k)
    %% Initalize Variables
    x = x0; % First approximate result
    r = abs(fun(x)); % First Residual
    it = 0; % Starting the Iterator
    xxs = [x0]; % Initalizing xxs with the first guess
    err = 1 + eps;  % giving a default error value bigger than eps

    %% Newton Method
    % Condition: err must be bigger than eps and maxIteration isn't reached yet
    while(err > eps && it < itMax)
        it = it + 1; % Increase iterator
        x = x - (fun(x)/dfun(x)); % Approximate x
        xxs = [xxs, x]; % Append approximated value to xxs
        err = abs(xxs(end) - xxs(end-1)); % Calculate absolute error difference
        r = abs(fun(x)); % Calculate new residual
    end
end
```

Multiple Roots

There are problem with multiple roots

- Function doesn't change sign at even multiple roots - can't bracket/bisection
- f(x) & f'(x) go to zero - possible division by zero (f(x) reaches zero faster but round off errors can occur)

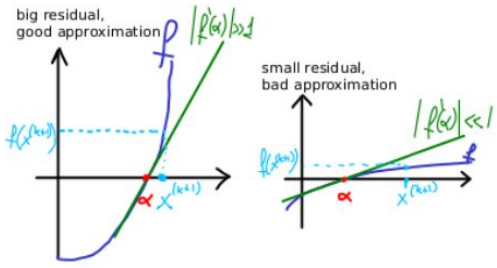
Secant Method

- This method doesn't require the derivative - derivative is approx. by the mean of a incremental ratio
- Is a **two point** method - requires two initial guesses
- Is quite efficient - just one function eval per step
- Can't use Newton but still faster than Bisection
- Instead of using the derivative it approximates it with a secant line
- Secant Method: $x_{n+1} = x_n - \frac{f(x_n)}{\left[\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}}\right]}$

```
function [x, r, it, xxs] =SecantMeth(fun, guess_1, guess_2, itMax, eps)
   %% INPUT:
   % fun           function handle,
   % guess_1       first guess
   % guess_2        second guess
   % itMax         number of maximum iterations
   % eps           prescribed convergence tolerance Îµ
   %% OUTPUT:
   % x             approximate result,
   % r             residual at the last iteration,
   % it            number of performed iterations,
   % eps           vector containing all computed x^(k)
   %% Initialize Variables
   x(1) = guess_1; % First approximate result
   x(2) = guess_2; % Second approximate result
   r = abs(fun(x)); % First Residual
   it = 2; % Starting the Iterator at 2
   err = 1 + eps;  % giving a default error value bigger than eps
   %% Secant Method
   while(err > eps && it < itMax)% Condition: err must be bigger than eps and maxIteration isn't reached yet
       it = it + 1; % Increase iterator
       % Approximate x with the Secant instead of derivative
       x(it) = x(it-1) - (f(x(it-1)))*((x(it-1) - x(it-2))/(f(x(it-1)) - f(x(it-2))));
       err = abs(x(end) - x(end-1)); % Calculate relative error - absolute difference
       r = abs(fun(x_1)); % Calculate new residual
   end
end
```

Stopping Criteria

- A convergent Newton method takes **infinite** iterations to reach exact root
- We require a termination or stopping criteria
- Use max. Number of iterations as good practice



Test on the Increment - checks absolute difference between two iterations
- Works well when f'(x) != 0

Test on Residual - |f(a)| = 0 - only works if |f'(x)| = ~1

Comparisons of Method

- Efficiency/velocity: (small) number of function evaluations (fast convergence).
- Easiness: easier methods are less prone to bugs.
- Robustness: fails rarely; if ever, it says clearly.
- Minimal amount of additional data required, such as the function's derivative.
- Minimal smoothness properties required to f.
- Possible easy generalization to many equations in many unknowns.

	Bisection	Newton	Secant
Velocity	✗	✓	✓
Easiness	✓	≈	≈
Robustness	✓	✗	✗
Need of additional data	✓	≈	✓
f behavior considered	✓	≈	≈
Generalization	✗	✓	✓

Fixed Point Iterations

- We look for x where f(x) = 0
- We can reformulate g(x) + x = x - Called Iterator Function

```
function [xs] = fixed_point(func, a, b, n)
   % Function that performs the fixed point iteration method to find the zeros of a function f given the associated
   % iteration function func.
   %   Args:
   %        func (function handle): the iteration function func (associated to the consired function f to find the zero for)
   %        a,b  (floats):          the bounds of the considered interval
   %        n    (integer):         the number of iterations to perform
   %   Returns
   %        xs   (float list):      the abscissa of the found zero of the function f
   %  Warning: the function does not tell you directly whether you converged or not

   % Creating a first guess and initialising the quantities
   x0 = a+0.8*(b-a)   % Setting the guess to be at 80% of the considered interval far from a
   xs=[x0];

   % Iterating a prescribed number of times
   % (for the sake of the examle we here consider a precise number of time step)
   for it = 1:n
       % Evaluating the value of the iterator function at the given point
       x1 = func(x0);
       % Updating the current value of the approximated zero
       x0 = x1;
       % Keeping track of the old values
       xs = [xs;x1];
   end
end
```

- With the Intermediate value theorem we can prove the existence of a fixed point in an interval

Banach Fixed Point Theorem

contraction mapping

If $\exists L < 1$ such that $\boxed{|\phi(x_1) - \phi(x_2)| \leq L|x_1 - x_2|, \quad \forall x_1, x_2 \in [a, b],}$

then there exists a **unique** fixed point $\alpha \in [a, b]$ of ϕ and the fixed point iterations **converges** to α, for any choice of $x^{(0)} \in [a, b]$.

3 - Polynomial Interpolation

What is Interpolation?

Interpolation is a special case of approximation that represents the data exactly at the given nodes.

Approximation of data - Data fitting.

- You're given a collection of data samples (x_i, y_i)
- X_i's are called nodes or abscissae
- Y_i's are called data values
- We want to find a function that describes the data continuously
    - So we can evaluate their value in the gaps
    - Or **Extrapolate** data outside the interval

Approximation of functions - Find a simpler function **v** for a complex function **f**.

- Same technique as in data fitting
- But we can choose the nodes e.g. data couples (x_i, y_i)

We only focus on interpolation in one dimension.

How to formulate the Problem

Problem formulation I

Notation:
$\{c_j\}_0^n$ **unknown coefficients**, or parameters,
$\{\Phi_j(x)\}_0^n$ **basis functions**, linearly independent.

Interpolant with a linear form

$$v(x) = c_0\Phi_0(x) + c_1\Phi_1(x) + \cdots + c_n\Phi_n(x) = \sum_{j=0}^{n} c_j\Phi_j(x)$$

- linearly independent $\Phi_j(x)$
  we have $v(x) = 0\ \forall x$ only if $c_j = 0$ for $j = 0, \ldots, n$.
  we can write ($n+1$ equations for $n+1$ unknowns)

$$\begin{bmatrix} \Phi_0(x_0) & \Phi_1(x_0) & \Phi_2(x_0) & \cdots & \Phi_n(x_0) \\ \Phi_0(x_1) & \Phi_1(x_1) & \Phi_2(x_1) & \cdots & \Phi_n(x_1) \\ \vdots & \vdots & \vdots & & \vdots \\ \Phi_0(x_n) & \Phi_1(x_n) & \Phi_2(x_n) & \cdots & \Phi_n(x_n) \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

- $\Phi_j(x)$ does not depend on y_i.
- $\Phi_j(x)$ simple and easy to be manipulated... any idea?

Construction of the interpolant - finding the coefficients for a given basis and set of data
Evaluation of the interpolant at a given point x

Polynomial Interpolation

Polynomial interpolation

- Simple to construct & evaluate
- Easy to manipulate (add, sum, differentiation)

Most importantly is their **UNIQUE** characteristic.

For n + 1 data points there is **one and only one** polynomial of degree less than or equal to n.

- There is only one straight line connecting two points
- There is only one parabola connecting 3 points
- …

But there are different approaches of computing those polynoms

Monomial Form

We construct the base functions in this way.

Monomial interpolation: $\Phi_j(x) = x^j$

$$\Pi_n(x) = c_0 + c_1 x + \cdots + c_n x^n = \sum_{j=0}^{n} c_j x^j$$

- + Easiest form, simple
- - System of n+1 linear eqs. in n+1 unknowns,
- - Matrix A can be ill-conditioned
- - Construction requires ⅔ n^3 flops
- + Evaluation cost is low, 2n flops
- - Coefficients cj are not indicative of f(x), change immediately if one value is modified

```
function [x] = MonomialMethod(nodes, y)
% x         coefficents
% nodes     x values
% y         evalued nodes

N = length(nodes)
A = ones(N, N)

for i=1:length(nodes)
    for j=1:length(nodes)
        A(i,j) = nodes(i)^(j-1)
    end
end

x = A \ y'
end
```

Lagrangian Interpolation

Opposite approach of monomial form, **coefficients are the data value**, then we look for suitable basis function.

Lagrangian form: $c_j = y_j$

$$\Pi_n(x) = \sum_{k=0}^{n} y_k \varphi_k(x), \quad \varphi_k(x) \in \mathbb{P}_n$$

$$\varphi_k(x_j) = \delta_{jk} = \begin{cases} 1 & j = k \\ 0 & \text{otherw.} \end{cases}$$

Lagrangian characteristic polynomials:

$$\varphi_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{(x - x_j)}{(x_k - x_j)}$$

The basis function $\Phi_j(x_i)$ is 1 if x_i equals the current index we're using to iterate over the basis functions.

Why does this work?
Whenever we have an x that equals one of our starting points all the other terms cancel out.

```
function [phi] = LagranCharPoly(nodes, x)
   %% Input Variables
   % nodes         vector of nodes
   % x             vector x
   %% Output Variables
   % phi           Matrix X x Nodes

   % Init Matrix Filled with Ones
   phi = ones(length(x), length(nodes));
   % Lagrange Polynomial Builder
   for i = 1:length(x) % for each row in phi
       for k=1:length(nodes) % for each column comp. in row i
           for j=1:length(nodes)
               if j ~= k
                   phi(i,k) = phi(i,k) * ((x(i)- nodes(j)) / (nodes(k) - nodes(j)));
               end
           end
       end
   end
end
```

- This just builds the phi Matrix

```
function [phi] = LagranianForm(nodesX, nodesY, x)
   %% Input Variables
   % nodes         vector of nodes
   % x             vector x
   %% Output Variables
   % phi           Matrix X x Nodes

   % Init Matrix Filled with Ones
   phi = ones(length(x), length(nodesX));
   % Lagrange Polynomial Builder
   for i = 1:length(x) % for each row in phi
       for k=1:length(nodesX) % for each column comp. in row i
           for j=1:length(nodesX)
               if j ~= k
                   phi(i,k) = phi(i,k) * ((x(i)- nodesX(j)) / (nodesX(k) - nodesX(j)));
               end
           end
       end
   end

   evaluatedPoints = sum(phi. * nodesY, 2); % sums by row keeping it a column vector

   plot(x, evaluatedPoints);
end
```

- This computes the Interpolated Polynomial over x

Barycentric Lagrange Interpolation

- The Lagrange Method can be rewritten in a way that it can
  - Be evaluated and updated in O(n)

$$p(x) = \sum_{j=0}^{n} f_j \ell_j(x), \qquad \ell_j(x) = \frac{\prod_{k=0,\, k\neq j}^{n}(x - x_k)}{\prod_{k=0,\, k\neq j}^{n}(x_j - x_k)}.$$

Let the Barycentric weights be defined as follows

Barycentric weights:

$$w_k = \left(\prod_{j=0, j\neq k}^{n} (x_k - x_j) \right)^{-1}$$

The numerator of lj can be rewritten

$$\ell(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$$

And then divide w_k by (x-x_k)

$$\Pi_n(x) = \ell(x) \sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k$$

To ensure that we cancel out the (x-xk) that is contained in l(x)

Stats

- O(n^2) flop to build all weight w_k
- 5n, O(n) to evaluate Polynomial P(x) at desired x
- O(n) to add a further node

CODEEE

Errors on polynomial Interpolation

Case: We can define the error

- If f(x) is differentiable
- F(x) has n+1 derivatives bounded in an interval we care about
- We can then quantify the difference of f and the polynomial for any point in the interval

Case: We can't define the error

- Use an error bound
- We compute an upper bound on $|f^{(n+1)}(x)|$ for $x \in I$
- And we maximize $|\prod_{i=0}^{n}(x - x_i)|$ for the given nodes in the interval

$$\max_{x\in I} |E_n f(x)| = \frac{1}{(n+1)!} \max_{x\in I} \left|f^{(n+1)}(x)\right| \max_{x\in I} \left|\prod_{i=0}^{n}(x - x_i)\right|$$

Equispaced Nodes

- Deriving an upper bound is easy
- Error estimate is valid for all x in Interval
- We can't deduce that error goes to zero by increasing the degree
  - You actually get weird oscillations
- The error is zero for x = x_i (as expected)
- If f is unknown and you w
- ant to estimate the error you can
  - Use a different subset of nodes to build a different polynomial and see how they match
  - If you have more than n+1 points use the additional ones for the approximation

Chebyshev Nodes

- Provide a good choice
- They minimize the error part $\prod(x - x_i)$.
- Clustered near endpoints of interval
- Yield good results
- Defined on a [-1, 1] interval but can be stretched to a wished interval

Chebyshev nodes
(or Chebyshev-Gauss nodes)

Defined for $x \in [-1, 1]$

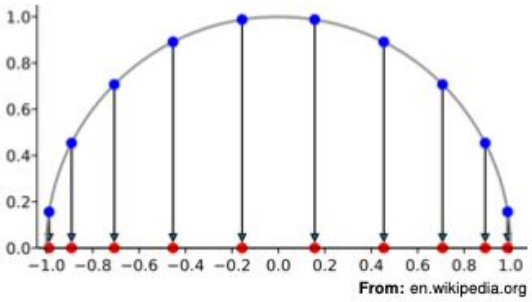
$$t_i = \cos\left(\frac{2i-1}{2n}\pi\right), \quad i = 1, \ldots, n$$

Affine transformation $[-1, 1] \to [a, b]$:

$x = a + \frac{b-a}{2}(t+1), \qquad t \in [-1, 1]$

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} t_i, \qquad x \in [a, b]$$

Example: $n = 10$



Chebyshev polynomials:
$T_k(x) = \cos(k \arccos(x))$

Stability of polynomial interpolation

Experiment using a polynomial f and crooked polynomial f'

- F' might be crooked due to rounding errors, uncertainty in data itself (measuring)

Polynomial interpolation is stable if, assuming the distance between f and f' is small, the difference between the two polynomials is small too. This happens if the **Lebesgue's constant** is small.

For large n, Lagrange interpolation on equispaced nodes can be unstable because the Lebesgue constant is large.

Lebesgue's constant

$$\Lambda_n(\mathbf{x}) = \max_{x\in[a,b]} \sum_{i=0}^{n} |\varphi_n(x)|$$

Stable for Λ_n small, but it depends on $\{x_i\}, i = 0, \ldots, n$

For Lagrange interpolation

- **on equidistributed nodes**

$$\Lambda_n \approx \frac{2^{n+1}}{e\,n\,(\log n + \gamma)} \leq \frac{2^{n+3}}{n}$$

- **on Chebyshev nodes**

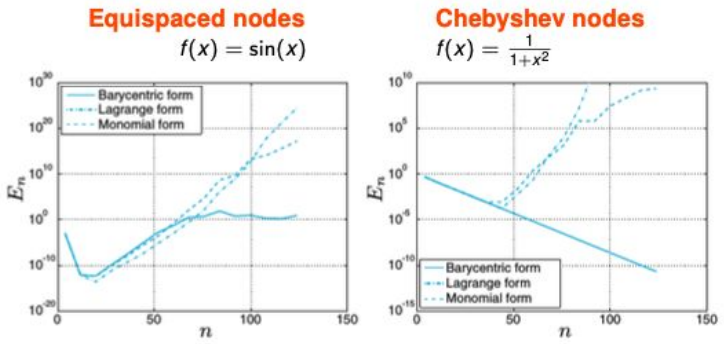
$$\Lambda_n \leq \frac{2}{\pi}\log(n+1) + \frac{2}{\pi}\left(\gamma + \log\frac{8}{\pi}\right) + \mathcal{O}\left(\frac{1}{n^2}\right)$$

with $e \approx 2.71834$, and $\gamma \approx 0.547721$.

For large n

- The Lebesgue constant for equispaced nodes grows exponentially with n
- For Chebyshev nodes it only grows logarithmically

In other words, when using Chebyshev nodes, the polynomial interpolation problem is well conditioned.



- Equispaced: increasing n, the error decreases for a while and then it increases, but while for Lagrange and monomial it diverges, for the barycentric form it keeps bounded after a certain point (n~65)
- Chebyshev: we do not have ill-conditioning, so increasing n, the error decreases as expected, until the round-off errors come into play, at least for Lagrange and monomial interpolations, which become are unstable, while barycentric form is stable also with respect to round-off error

4 - Piecewise polynomial Interpolation

Piecewise polynomial interpolation

Problem with Polynomial interpolation, when **n** is large

- P_n(x) may become oscillatory
- P_n(x) is differentiable up to a high-order (even infinitely differentiable, but data are usually only piecewise smooth

Solution Idea

- Combine more interpolants
- Apply lower-order polynomials to subsets of data

Piecewise polynomial interpolation

Locally: low degree polynomial in each subinterval
Globally: Patch all polynomials together to form a continuous global interpolating curve

Linear piecewise interpolation

Basically “Connect the dots”

For each interval we use the slope function to **connect the dots**.

For an arbitrary x in the data pair range

1. Find the interval where it’s included
2. Apply Slope Function for your desired value X
3. for $x \in I_i$:

$$s_i(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i)$$

s_i is the $\Pi_1(x)$ of nodes $[x_i, x_{i+1}]$.

There are **no** new extremas, minimum and maximum values are preserved

Matlab Command: *interp1(x,y,z)*

Error

Error of polynomial interpolant of degree 1 for a function with a bounded interval is

$$E_1 f(x) = f(x) - \Pi_1 f(x) = \frac{f''(\xi)}{2}(x - x_i)(x - x_{i+1}),$$

Where ξ is any value in the current Interval.
We can calculate an upper bound.

$$|f(x) - \Pi_1^H f(x)| \leq \frac{H^2}{8} \max_{\xi \in I} |f''(\xi)|, \qquad \text{where} \quad H = \max_{1 \leq i \leq n} (x_i - x_{i-1})$$

Problem - not smooth enough, it is desirable to have at least continuous derivatives
Slope is not the same at the nodes - not differentiable

```
function [x_space, y_space] = PiecewiseLinearInterpolation (nodesX, nodesY)
    %% Input
    % nodesX = [3.0, 4.5, 7.0, 9.0]     Datapoints X
    % nodesY = [2.5, 1.0, 2.5, 0.5]     Datapoints Y
    %%

    x_space = [] % We create for each interval x values to be evaluated
    y_space = []

    for i = 1:length(nodesX)-1
        x = linspace(nodesX(i),nodesX(i+1),100)
        x_space = [x_space, x]
        y = nodesY(i) + (nodesY(i+1)-nodesY(i))/(nodesX(i+1)-nodesX(i)) * (x - nodesX(i)) % This is the slope formula
        y_space = [y_space, y]
    end


    plot(nodesX,nodesY,'bo',x_space,y_space,'r-')
    title 'Linear Piecewise Polynomial Interpolation'
    xlabel X
    ylabel Y
end
```

Approximation by Spline Function

We want

- Piecewise polynomial interpolation for a degree m >= 2
- Ensure that (m-1) derivatives are continuous at the knots

A **spline** of at least *m* order has to be used

Piecewise constant interpolation - we don't use the nodes as our break points but the midpoint of the intervals.

- In each sub-interval the approximation equals the value at the midpoint of the current interval

Error of piecewise constant interpolation is bounded:

$$|f(x) - \Pi_0^H f(x)| \leq H \max_{\xi \in I} |f'(\xi)|.$$

Quadratic Splines

- Interpolation using second order polynomials
- Continuous first derivatives at the nodes
- In addition to the interpolating conditions and the derivative continuity, we need a further condition to have the **same number of equations and unknowns.**
    - Second derivative at x_0 should be 0

2 n Equations
c_1 * x_0^2 + b_1* x_0 + a_1 = y_0
c_1 * x_1^2 + b_1 * x_1 + a_2 = y_1

N-1 Equations
2 c_1 x_1 + b_1 - 2 c_2 x_1 - b_2 = 0

+ 1 Equation
2 c_1 * x_0^2 + b_1 = 0

- C_1 = 0

As part of solving this we want to build a coefficient matrix - coefficients as rows and subintervals as columns

We are looking for si(x) = ai +bi(x−xi)+ci(x−xi)^2 for i = 0,1,...,n−1, so we have 3n unknowns (n polynomials with 3 unknown coefficient each).

We need 3n conditions/equations:

- N conditions: At the left border we have a_i = y_i
    - Due to x-xi being zero
- N conditions: At the right border we have $b_i + c_i h_i = (y_{i+1} - y_i)/h_i$ for $i = 0, 1, \ldots, n-1.$
- N-1 conditions: Continuity of first derivative at nodes
- Additional Condition second order derivative at x_0 = 0

Computing By Hand
From this formula
si(x) = ai +bi(x−xi)+ci(x−xi)^2 for i = 0,1,...,n−1
We can derive:

- a_i = y_i
- b_i = (y_i+1 - y_i) / h_i - c_i*h_i hi = x_i+1 - x_i
- c_0 = 0
- c_i h_1 = -c_i-1 * h_i-1 + (y_i+1 - y_i) / h_i - (y_i - y_i-1) / h_i-1

```
% Defining the spline nodes
nodes = [0.25, 1.7057;
        0.75, 2.7449;
        1.3,  2.5580;
        1.7,  3.0326;
        1.9,  7.8285];

% Definition of the testing vector (where the path will be rebuilt)
x0 = 0.25; xn = 1.9;
xtest = linspace(x0, xn, 100);

%% Interpolating the quadratic spline
coeff = QuadraticSpline_Hand(nodes);% Retrieving the coefficients of the quadratic spline
ytestmat = Interpolating_Hand(nodes, xtest); % Retrieving the interpolating path

%% Plotting the reconstructed path and the sticks
fig = figure();
hold on
plot(ytestmat(:,1), ytestmat(:,2), "--r")
plot(nodes(:,1), nodes(:,2), "or")
hold off
legend("Reconstructed path", "Nodes")
title("Path plot")
xlabel("x")
ylabel("y")

function [interppoints] = Interpolating_Hand(xx, xtest)
   % Initialising the interpolated vector
   ytest = [];
   xtt   = [];
   % Getting the interpolation coefficients from the nodes
   coeff = QuadraticSpline_Hand(xx);
   % Evaluate the points values depending on in which interval they lie
   for inter = 1:(length(xx)-1)
           % Retrieving the points living in the interval inter-1
           index = find((xx(inter+1)>=xtest) .* (xx(inter)<=xtest)); % we need this to know which row of the coefficent to use
           if ~isempty(index)
               % Tabulating the values
               xtemp = xtest(index)-xx(inter);
               % Constructing the determination matrix and retrieving the
               % interpolated values
               A = [ ones(1, length(index)); xtemp; xtemp.^2];
               values = coeff(inter,:)*A;
               % Reconstructing the values within this interval
               ytest = [ytest, values];
               xtt   = [xtt, xtest(index)];
           end
   end
   % Returning the interpolated xy valyes
   interppoints = [xtt; ytest]';
end

function [coeff] = QuadraticSpline_Hand(nodes)
   % Function computing the coefficients of the quadratic spline in
   % the form si(x) = ai+bi(x-xi)+ci(x-xi)^2
   % Input:    (nx2 floats):  the nodes (abscissa and values) to be used to build the quadratic interpolation
   % Output:   (n x 3 floats):  the matrix containing the coefficients [c0 b0 a0]  associated to each node on a row

   % Retrieving the number of given nodes
   n = length(nodes);
   % Computing the horizontal gap vector | x_i+1 - x_i
   h = nodes(2:end,1)-nodes(1:end-1,1);

   %% Computing the coefficients ai | a_i = y_i
   a = nodes(1:end-1,2);

   %% Computing the coefficients  bi and ci
   c = zeros(n-1,1); % we already get c_0 = 0 here
   for k = 2:(n-1)
       c(k) = (1./h(k)).*((nodes(k+1,2)-nodes(k,2))./h(k)-(nodes(k,2)-nodes(k-1,2))./h(k-1)-c(k-1).*h(k-1));
   end

   % Computing b knowing c
   b = (nodes(2:end,2)-nodes(1:end-1,2)) ./ h - c.*h;

   %% Concatenating the coefficients in  the matrix coeff
   coeff = [a'; b'; c']';
end
```

Computing by Linear Solver

We can also solve this by setting up one big linear system to be solved.

$$
\begin{pmatrix}
1 & x_0 & x_0^2 & & & & & \\
1 & x_1 & x_1^2 & & & & & \\
 & & & 1 & x_1 & x_1^2 & & \\
 & & & 1 & x_2 & x_2^2 & & \\
 & & & & & & \ddots & \\
0 & 1 & 2x_0 & 0 & -1 & -2x_1 & & \\
 & & & 0 & 1 & 2x_1 & 0 & -1 & -2x_2 \\
0 & 0 & 1 & 0 & \cdots & & & &
\end{pmatrix}
\cdot
\begin{pmatrix}
a_0 \\ b_0 \\ c_0 \\ a_1 \\ b_1 \\ c_1 \\ \vdots
\end{pmatrix}
=
\begin{pmatrix}
y_0 \\ y_1 \\ y_1 \\ y_2 \\ y_2 \\ \vdots \\ 0 \\ 0 \\ 0
\end{pmatrix}
$$

- ensures $c_0 = 0$
- ensures $b_0 + 2c_0x_0 - b_1 - 2c_1x_1 = 0$
- ensures $a_0 + b_0x_0 + c_0x_0^2 = y_0$

  $a_0 + b_0x_1 + c_0x_1^2 = y_1$

Code see exercise 6 - it’s a pain

Cubic Splines

- Use third order polynomials
- Ensure continuous first and second order derivatives

Conditions:

- Interpolating condition
- Condition of Continuity of first and second derivatives
- Two more are left
    - Imposing zero second order derivatives at x0 and x1
    - “Complete Spline” but you need the derivative function -> bad to fit data
    - “Not a knot spline” first derivative of underlying function -> not useful to fit data

By manipulating equations you can get a tridiagonal system for coefficients c_i

- A linear system with non zero matrix entries
- Along the main diagonal and super and sub diagonal
- Only including c_i-1, c_i, c_i+1

Compute 4n coefficients with this formula

$$s_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad \text{for } i = 0, 1, \ldots, n-1\,.$$

We approach this problem by starting with c and building d,b,a from there

Following these tips:

- combine the interpolating conditions $s_i(x_i) = y_i$ and $s_i(x_{i+1}) = y_{i+1}$ to have a relation that includes only the coefficients b_i, c_i and d_i and the values y_i and y_{i+1}.
- use the condition $s_i''(x_{i+1}) = s_{i+1}''(x_{i+1})$ to find a definition of d_i in terms of coefficients c_i and c_{i+1}. Derive the analogue expressions for d_{i-1}.
- plug the result of the previous step into the expression obtained in the first step, to have an expression of b_i in terms of only the coefficients c_i and c_{i+1} and the values y_i and y_{i+1}. Derive the analogue expressions for b_{i-1}.
- substitute the expressions found above into the condition $s_{i-1}'(x_i) = s_i'(x_i)$ to achieve an expression including only c coefficients.

You arrive at an recurrence expression like this

$$\frac{1}{3}h_{n-2}c_{n-2} + \frac{2}{3}\left(h_{n-2} + h_{n-1}\right)c_{n-1} = \frac{y_n - y_{n-1}}{h_{n-1}} - \frac{y_{n-1} - y_{n-2}}{h_{n-2}}$$

- Assuming c_0 = 0

```
function [coeff] = CubicSpline(nodes,yys)
   % Method to perform Natural Spline Coeficents Creation
   % Arguments:
   %   nodes - n+1 number of nodes
   %   yys - n+1 values for corresponding to nodes
   % Outputs:
   %   coeff - coefficent matrix of size n * 4
   n=length(nodes);
   a=zeros(n-1,1);
   b=zeros(n-1,1);
   AA=zeros(n-2,n-2); % intermediate step matrix for c vector
   d=zeros(n-1,1);
   %% Step Size Creation - Computing the horizontal gap vector
   h = nodes(2:end)-nodes(1:end-1);
   %% Setting up AA Matrix to get c
   % Assigning Default Values
   AA(1,1)=2/3 *h(1)^2 + h(2)-1/3*h(2)^2;
   AA(1,2)=1/3*h(2)^2;
   % Populating n-2 row entries
   AA(n-2, n-3) = h(n-3)-2/3*h(n-3)^2;
   AA(n-2,n-2) = 2/3*h(n-3)^2 + h(n-2)-1/3*h(n-2)^2;
   % Assinging Values based on Formulas
   for i=2:n-3
       AA(i,i-1)= h(i-1)-2/3*h(i-1)^2;
       AA(i,i)= 2/3*h(i-1)^2 + h(i)-1/3*h(i)^2;
       AA(i,i+1)= 1/3*h(i)^2;
   end
   for i=2:length(yys)-1
       y(i-1)=(yys(i+1)-yys(i))/h(i)-(yys(i)-yys(i-1))/h(i-1);
   end

   c = AA\y';
   c = [0;c]; %% appending empty 0 as first entry
   %% Calculate d
   for i=1:n-2
       d(i)=1/3*(c(i+1)-c(i));
   end
   d(n-1)=1/3*(-c(n-1));
   %% Calculate b
   for i=1:n-1
       b(i)=(yys(i+1)-yys(i))/h(i) - c(i)*h(i)-d(i)*h(i)^2;
   end
   %% Calculate a
   for i=1:n-1
       a(i)=yys(i);
   end
   %% Return coefficent
   coeff=[d c b a];
end
```

Error on cubic spline

Error of cubic spline For the complete spline approximation, we have an estimate also for the derivatives: it performs pretty well also for derivatives. We have a similar estimate also for the not-a-knot condition, while the natural spline is generally only second order accurate near the endpoints.

Hermite piecewise cubic interpolation

Quadratic and Cubic spline **do not preserve monotonicity** - may create new min and max values

Accomplished by imposing **interpolating condition for the first derivative of f** at nodes - 2n conditions (no special single conditions needed)

MatLab Command **pchip**

COOODE

5 - Least Squares & Review Linear Algebra

Linear Least Squares

Given a set of data find a model function f~(x) that approximates the trend without interpolating

- Model function contains few parameters
- Data Fitting: compute the parameters on the basis of the data
- Finds the “best” approximation

Find the minimal possible value of the sum of squares of the residuals

Residual is the difference between the estimated and the “real” value

```
function [coeff] = leastSquares (xxs, yys, m)
    %% Input
    %   xxs     x value of data nodes
    %   yys     y value of data nodes
    %   m       degree of polynomial approximating data
    %% Output
    %   coeff [a0, a1, ..., an]

    yValues = yys'; % make it a column

    %% Construct Matrix A
    % [1, x_1^1, x_1^2;
    %  1, x_2^1, x_2^2 ]
    a = [];
    for row=1:length(xxs)
        currentRow = [];
        currentM = m;
        for i=1:m+1
            currentRow = [currentRow, xxs(row)^currentM];
            currentM = currentM - 1;
        end
        a = [a; currentRow];
    end
    coeff = a\yValues;
end
```

Linear Algebra Review - Read Summary 5

Polynomial Least-squares

For $\{(x_i, y_i)\}, i = 1, \ldots, n$, for a given $m \geq 1$ (usually, $m \ll n$), we look for a polynomial $\tilde{f} \in \mathbb{P}_m$ which satisfies, for every polynomial $p_m \in \mathbb{P}_m$,

$$\sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \left[y_i - \tilde{f}(x_i)\right]^2 \leq \sum_{i=1}^{n} [y_i - p_m(x_i)]^2$$

If it exists, $\tilde{f}$ is the **least-squares approximation** in $\mathbb{P}_m$.

Function shape: $\tilde{f}(x) = a_0 + a_1 x + \cdots + a_m x^m = \sum_{j=0}^{m} a_j x^j$

Problem to be solved:
Find $a_0, a_1, \ldots, a_m$ such that

$$\Psi(a_0, a_1, \ldots, a_m) = \min_{b_i, i=0,\ldots,m} \Psi(b_0, b_1, \ldots, b_m)$$

where $\Psi(b_0, b_1, \ldots, b_m)$ is the sum of squared residuals:

$$\Psi(b_0, .., b_m) = \sum_{i=1}^{n} \Big[\underbrace{y_i - (b_0 + b_1 x_i + .. + b_m x_i^m)}_{\text{residual } r_i}\Big]^2$$

Model function: $\tilde{f}(x) = \sum_{j=0}^{m} a_j x^j$ $\qquad$ $\Psi(a_0, \ldots, a_m) = \sum_{i=1}^{n} \left[y_i - \sum_{j=0}^{m} a_j x_i^j\right]^2$

Taking the derivative we get the following (we trick by scaling by 0.5 - minimizer stays the same anyways)

$$\frac{\partial \Psi}{\partial a_k} = \sum_{i=1}^{n} \left[\left(y_i - \sum_{j=0}^{m} a_j x_i^j\right)\left(-x_i^k\right)\right] = 0 \quad \text{for } k = 0, \ldots, m$$

or

$$\sum_{i=1}^{n} \left[x_i^k \left(\sum_{j=0}^{m} a_j x_i^j\right)\right] = \sum_{i=1}^{n} \left(y_i x_i^k\right) \quad \text{for } k = 0, \ldots, m$$

By transposing rows to have columns that all share the same exponent we can simplify things. We get a linear system.

Normal equations:

$$B^{\mathrm{T}} B \mathbf{a} = B^{\mathrm{T}} \mathbf{y}$$

$$B = \begin{bmatrix} 1 & x_1 & \ldots & x_1^m \\ \ldots & \ldots & \ldots & \ldots \\ 1 & x_i & \ldots & x_i^m \\ \ldots & \ldots & \ldots & \ldots \\ 1 & x_n & \ldots & x_n^m \end{bmatrix} \qquad \mathbf{a} = \begin{bmatrix} a_0 \\ \vdots \\ a_m \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ \ldots \\ y_n \end{bmatrix}$$

$B \in \mathbb{R}^{n \times (m+1)}$ Vandermonde matrix $\qquad$ $\mathbf{a} \in \mathbb{R}^{m+1}$ coefficients $\qquad$ $\mathbf{y} \in \mathbb{R}^n$ values

- $B^{\mathrm{T}} B \in \mathbb{R}^{(m+1) \times (m+1)}$ we have a linear system!

If n == m we have polynomial interpolation - *polyfit(x,y,m)*

B\y = a

Linear Least-squares

Linear Regression - finding a straight line approximate (in the least square way) for the data

Model function: $\tilde{f}(x) = a_0 + a_1 x$

Expanding our Model Function a bit gives us this

Model function: $\tilde{f}(x) = a_0 + a_1 x$

$$\begin{aligned}\Psi(a_0, a_1) &= \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i)]^2 \\ &= \sum_{i=1}^{n} \left[y_i^2 + (a_0 + a_1 x_i)^2 - 2(a_0 + a_1 x_i) y_i\right] \\ &= \sum_{i=1}^{n} \left[y_i^2 + a_0^2 + a_1^2 x_i^2 + 2a_0 a_1 x_i - 2a_0 y_i - 2a_1 x_i y_i)\right]\end{aligned}$$

We know will try to minimize the term by computing the derivative respective to the two unknowns and make them equal to zero. (Partial derivative)

$$\frac{\partial \Psi}{\partial a_0}(a_0, a_1) = 0 \qquad \frac{\partial \Psi}{\partial a_1}(a_0, a_1) = 0$$

$$\frac{\partial \Psi}{\partial a_0} = \textstyle\sum_{i=1}^{n} [2a_0 + 2a_1 x_i - 2y_i)] \qquad \longrightarrow \textstyle\sum_{i=1}^{n} [a_0 + a_1 x_i - y_i)] = 0$$

$$\frac{\partial \Psi}{\partial a_1} = \textstyle\sum_{i=1}^{n} \left[2a_1 x_i^2 + 2a_0 x_i - 2x_i y_i)\right] \qquad \longrightarrow \textstyle\sum_{i=1}^{n} \left[a_0 x_i + a_1 x_i^2 - x_i y_i)\right] = 0$$

Removing the scalars doesn't matter for the minimization. You get the normal equations for m=1.

$$\sum_{i=1}^{n} [a_0 + a_1 x_i - y_i)] = 0 \qquad \longrightarrow \qquad (n)a_0 + a_1 \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$$

$$\sum_{i=1}^{n} \left[a_0 x_i + a_1 x_i^2 - x_i y_i)\right] = 0 \qquad \longrightarrow \qquad a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i$$

$\hookrightarrow$ Normal equations (here for $m = 1$)

6 - Direct Methods for Linear Systems

Linear Systems often appear as part of a subproblem or a standalone problem
We want to solve it directly!

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$\boxed{\begin{array}{c} A\mathbf{x} = \mathbf{b} \\ [A]\{x\} = \{b\} \end{array}}$$

solvable iff A is nonsingular

Note: iff = "if and only if", or $\Leftrightarrow$

or, equivalently:

- $\det(A) \neq 0$
- A^{-1} exist, such that $A^{-1}A = AA^{-1} = I$
- Columns (or rows) of A linearly independent.

Cramer's Rule

To solve A**x** = **b**

- We calculate the determinant of A
- Foreach x_i component we replace the ith column of the matrix A with the b vector and calculate the determinant
- We divide this modified det(A_i) / det(A) to get the x_i component
- Super impractical for large n

Cramer rule: solution of $A\mathbf{x} = \mathbf{b}$

$$x_i = \frac{\det(A_i)}{\det(A)}, \quad i = 1, \dots, n$$

Example: $n = 3, i = 2$

$$x_2 = \frac{\det \begin{bmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{bmatrix}}{\det(A)}$$

```
function [x] = CramersRule(A,b)
    %% Input
    %   A   nonsingular Matrix A
    %   b   result vector b
    %%Output
    %   x   Solution to A*x = b
    %% Method
    [rownum,colnum]=size(A)
    x = []
    detA = det(A)
    for i=1:colnum
        copyA = A;
        copyA(:,i) = b
        x_i = det(copyA)/detA
        x = [x,x_i]
    end
end
```

Backward Substitution

We want to solve an **upper triangular system**

- U * **x = b** - where U is **upper triangular**, **non-singular**
- U_ii can't be 0 - diagonal can't have zeros in it

Complexity: n^2 operations

For each i>= 2 we need $(n-i)+(n-i)+1 = 2(n-i)+1$ operations

Repeating for each i $\sum_{i=1}^{n}[2(n-i)+1] = 2\sum_{i=1}^{n-1} i + \sum_{i=1} 1 = n(n-1)+n$ **= n^2**

Algorithm: Backward substitution
INPUT: upper triangular matrix U, vector **b**

FOR i going from n to 1 (increment -1):

$$x_i = \frac{1}{u_{ii}}\left(b_i - \sum_{j=i+1}^{n} u_{ij}x_j\right)$$

END
OUTPUT: vector **x**.

triu(A)

```
function [x] = BackwardSubstitution(U,b)
    %% Input
    %   U   Upper triangular Matrix - is nonsingular - no zeros in diagonal
    %   b   Result Vector
    %% Output
    %   x   Result for U*x = b
    %% Code
    n = size(b,2)
    x = zeros(1,n)

    for i=n:-1:1
            xi= 1/U(i,i) .* (b(i)- sum( U(i,i+1:end) .*x (i+1:end) ) );
            x(i)=xi;
    end
    x=x';
end
```

Forward Substitution

We want to solve an **lower triangular system**

- L * **x = b** - where L is **lower triangular**, **non-singular**
- L_ii can't be 0 - diagonal can't have zeros in it

Complexity: n^2 operations

For each i>= 2 we need $(i-1)+(i-2)+2 = 2i-1$ operations

Repeating for each i $\sum_{i=1}^{n}(2i-1) = 2\sum_{i=1}^{n} i - \sum_{i=1} 1 = n(n+1) - n = n^2$ **= n^2**

tril(A)

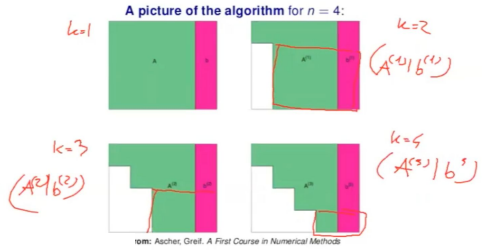
Algorithm: Forward substitution
INPUT: upper triangular matrix L, vector **b**
FOR i going from 1 to n:

$$x_i = \frac{1}{\ell_{ii}}\left(b_i - \sum_{j=1}^{i-1} \ell_{ij} x_j\right)$$

END
OUTPUT: vector **x**.

```
function [x] = ForwardSubstitution(L,b)
    %% Input
    %   L   Lower triangular Matrix - is nonsingular - no zeros in diagonal
    %   b   Result Vector
    %% Output
    %   x   Result for L*x = b
    %% Code
    n = size(b,2)
    x = zeros(1,n)

    for i=1:n
            xi= 1/L(i,i) .* (b(i)- sum( L(i,1:i-1) .*x (1:i-1) ) );
            x(i)=xi;
    end
    x=x';
end
```

Gaussian Elimination



Solution of A**x** = **b** is **invariant** to

- Multiplication of a row by a constant
- Subtraction of a multiple of one row from another

Example: Start from...

$$\left[\begin{array}{rrr|r} 2 & 1 & -1 & 8 \\ -3 & -1 & 2 & -11 \\ -2 & 1 & 2 & -3 \end{array}\right]$$

performs some of the above modifications,

and then solve

$$\left[\begin{array}{rrr|r} 1 & a'_{12} & a'_{13} & b'_1 \\ 0 & 1 & a'_{23} & b'_2 \\ 0 & 0 & 1 & b'_3 \end{array}\right]$$

with backward substitution!

Notation: $(A|\mathbf{b})$ augmented matrix

- Row interchanges

Use these row transformations to transform the general linear system in an equivalent upper triangular form.

(Forward) Gaussian Elimination

INPUT: square matrix A, vector **b**

FOR $k = 1, \ldots, n-1$	*loop on diagonal elements*
FOR $i = k+1, \ldots, n$	*loop on rows*
$l_{ik} = \dfrac{a_{ik}}{a_{kk}}$	$a_{kk} \neq 0$ *pivot elements*
FOR $j = k+1, \ldots, n$	*loop on columns*
$a_{ij} = a_{ij} - l_{ik} a_{kj}$	*re-write A, not store 0*
END	
$b_i = b_i - l_{ik} b_k$	*re-write* **b**
END	
END	

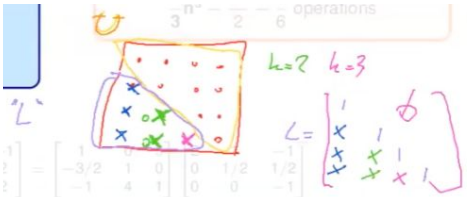
OUTPUT: upper triangular part of modified A, and modified **b**.

Gauss Elimination
Complexity: $\frac{2}{3} n^3$ operations

```
function x = GaussJordan (A,b)
   % Solve the problem Ax=b using Gauss-Jordan method
   % We suppose that no pivoting is required
   % INPUT :   A:   Coefficient matrix
   %           b:   Right-hand term
   %
   % OUPUT :   x:   Solution

   % Error message if necessary
   [n,m]=size(A);
   b=b(:); %Column vector

   % Concatenation
   A=[A b];

   % Reduction
   for j=1:n %Go accross columns (except last: it is b) : each pivot
     for i=1:n %Accross rows: goal: eliminate the rest of the column
        if i~=j %Reduction on all rows except itself
           factor=A(i,j)/A(j,j);
   %          A(i,:)=A(i,:)-A(j,:)*A(i,j)/A(j,j);
           for k=1:(n+1)
              A(i,k)=A(i,k)-A(j,k)*factor;
           end
        end
     end
   end

   % Debug A

   % Init output
   x=0.*b;

   % Retrieve modified b
   b=A(:,end);


   % Final solving
   for i=1:n
      x(i)=b(i)/A(i,i);
   end
end


A=eye(3);b=(1:3)';
GaussJordan(A,b)
A\b
fprintf("\n")
```

```
function x = GaussElimination(A,b)

    % Error message if necessary
    [n,m]=size(A);
    b=b(:); %Column vector

    %% Triangularisation
    for j=1:(n-1)
        for i=(j+1):n
            b(i)=b(i)-b(j).*A(i,j)./A(j,j);
            A(i,:)=A(i,:)-A(j,:).*A(i,j)./A(j,j);
        end
    end

    disp("Upper triangular matrix:")
    disp(A)

    %% Backward solving

    for j=n:-1:1
        b(j)=b(j)./A(j,j);
        for i=1:(j-1)
            b(i)=b(i)-b(j)*A(i,j);
        end
    end

    x=b;
end

A= [1 -1 3;
    1 1  0;
    3 -2 1];
b=[2;4;1];

disp("Gaussian elimination result")
GaussElimination(A,b)
disp("Embedded Matlab backslash")
A\b
```

LU Factorization



If you're given any Matrix A there exists

- L lower triangular
- U upper triangular

Such that A = L*U

Create a list of operations to operate on our new vector b without recomputing to check other b's. Improved Gaussian Elimination. Instead of applying the factor l_ik in the Gaussian Elimination Method we save the factor in the Matrix A at the same position.

- L Matrix stores the factors and when going through the algorithm, once you store the factors they aren't edited anymore we move to the next column and down one row, ….
- U is our upper matrix with the modified values
- We artificially add the 1 diagonal to L for completion reasons

If A is non singular so is L & U

We split the solution of Ax = b -> LUx = b

- 1. Ly = b - forward substitution
- 2. Ux = y - backward substitution

Number of Equations & Number of Unknowns
n x n unknowns + n (because we have the diagonal twice!)
n x n equations

Gauss LU factorization

INPUT: square matrix A

FOR $k = 1, \ldots, n-1$
  FOR $i = k+1, \ldots, n$
    $a_{ik} = \frac{a_{ik}}{a_{kk}}$
    FOR $j = k+1, \ldots, n$
      $a_{ij} = a_{ij} - l_{ik} a_{kj}$
    END
  END
END

OUTPUT: Modified A.

At the end:

- $U_{ij} = a_{ij}$ $i = 1, \ldots, n;$ $j = i, \ldots, n,$
- $L_{ij} = a_{ij}$ $i = 2, \ldots, n;$ $j = 1, \ldots, i-1.$

MATLAB: `tril`, `triu`

Complexity:
$\frac{2}{3}n^3 - \frac{n^2}{2} - \frac{n}{6}$ operations

This imbalance can be solved by **defaulting L Matrix Diagonal to 1** or **adding n equations** l_ii = 0

Usage of Gaussian elimination: Ax = b
Given A real, non-singular Matrix, **perform LU decomposition** - most computational demanding step (cubic)
A = LU
Given any right hand side vector **b**

1. Solve **Forward Substitution** L**y** = **b** - take your computer y - (quadratic)
2. Solve **Backward Substitution** U**x** = **y** - (quadratic)

You have your vector **x** you were looking for - Det(L) = 1 | Det(A) == Det(U)

```
function [L,U] = LUDecomposition(A)
    %% INPUT
    % A     nonsingular, quadratic Matrix A, with no zeros please
    %% OUTPUT
    % L     lower triangular matrix with multiplication factor and a
    % diagonal full of ones
    % U     upper triangular matrix with modified values from A
    %%
    n = size(A, 1); % Obtain number of rows or columns (quadratic!)

    % Default 1 as diagonal
    L = eye(n); % Start L off as identity and populate the lower triangular half slowly

    for k = 1 : n
        % For each row k, access columns from k+1 to the end and divide by
        % the diagonal coefficient at A(k ,k)
        L(k + 1 : n, k) = A(k + 1 : n, k) / A(k, k);

        % For each row k+1 to the end, perform Gaussian elimination
        % In the end, A will contain U
        for j = k + 1 : n
            A(j, :) = A(j, :) - L(j, k) * A(k, :);
        end
    end
    U = A;
end
```

Pivoting

If we encounter zeros we can run into errors where we divide by zero e.g. not good

We can switch rows to circumvent these, any row changes should be reflected in a Permutation Matrix
Permutation Matrix P - Initially Identity matrix - copies permutations done on A

Gauss LU factorization with p. p.
INPUT: square matrix A

Initialize $P = \mathbb{I}_n$
FOR $k = 1, \ldots, n-1$
  FIND $q \geq k : |a_{qk}| = \max_{k \leq i \leq n} |a_{ik}|$
  Exchange rows k and q in A and P

  FOR $i = k+1, \ldots, n$
    $a_{ik} = \dfrac{a_{ik}}{a_{kk}}$
    FOR $j = k+1, \ldots, n$
      $a_{ij} = a_{ij} - l_{ik} a_{kj}$
    END
  END
END

OUTPUT: Modified A, P.

Solving Systems - PA = LU
When solving Ax = b

L**y** = P**b**
U**x** = **y**

Round off errors are amplified and accumulated if $|a_{ii}^{(k)}| \approx 0$

```
function [L,U, P] = LUDecompositionPivot(A)
    % LU Decomp. with Partial Pivoting
    %% INPUT
    % A     nonsingular, quadratic Matrix A, with no zeros please
    %% OUTPUT
    % L     lower triangular matrix with multiplication factor and a
    % diagonal full of ones
    % U     upper triangular matrix with modified values from A
    % P     Identity Matrix with shifted row information
    %%
    n = size(A, 1); % Obtain number of rows or columns (quadratic!)
    P = eye(n);
    U = zeros(n);
    % Default 1 as diagonal
    L = zeros(n); % Start L off as identity and populate the lower triangular half slowly

    for k = 1 : n
        % find the entry in the left column with the largest abs value (pivot)
        [~,r] = max(abs(A(k:end,k)));
        r = n-(n-k+1)+r;

        A([k r],:) = A([r k],:);
        P([k r],:) = P([r k],:);
        L([k r],:) = L([r k],:);

          % from the pivot down divide by the pivot
        L(k:n,k) = A(k:n,k) / A(k,k);

        U(k,1:n) = A(k,1:n);
        A(k+1:n,1:n) = A(k+1:n,1:n) - L(k+1:n,k)*A(k,1:n);
    end
    U = A;
end
```

The Condition Number

Well-conditioned system: a small change of a value in A -> results in a small change in x
Ill-conditioned system: a small change of a value in A -> results in a large change in x

Ill conditioned problems are extremely sensitive to round-off error

Hilbert Matrix

Hilbert matrix: $A_n \in \mathbb{R}^{n \times n}$

$$a_{ij} = \frac{1}{i+j-1} \quad i,j = 1,\dots,n$$

$$A_n = \begin{bmatrix} 1 & 1/2 & 1/3 & \dots & 1/n \\ 1/2 & 1/3 & 1/4 & \dots & 1/(n+1) \\ 1/3 & 1/4 & 1/5 & \dots & 1/(n+2) \\ \dots & \dots & \dots & \dots & \dots \\ 1/n & 1/(n+1) & 1/(n+2) & \dots & 1/(2n-1) \end{bmatrix}$$

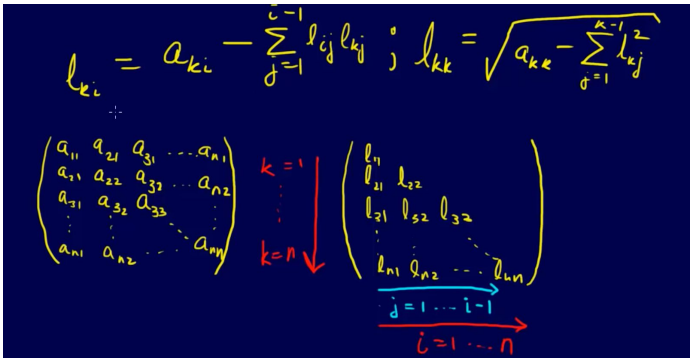
How accurate is the solution of direct methods?

Cholesky factorization

An efficient factorization for symmetric positive definite matrices - if A is **symmetric positive definite**

$$\boxed{\text{Cholesky factorization: } A = R^{\mathrm{T}} R}$$

We can divide A into L & U - A = LU; If A = A^T we can divide A = L*L^t



Cholesky factorization

INPUT: square matrix A

Initialize R

$r_{11} = \sqrt{a_{11}}$

FOR $j = 2, \dots, n$

  FOR $i = 1, \dots, j-1$

$$r_{ij} = \frac{1}{r_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj} \right)$$

  END

$$r_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2}$$

END

OUTPUT: Upper triangular R.

Cost: ⅓ n^3 operations
Matlab: R = chol(A)

```
function [R] = Cholesky(A)
    %% Input - A (symmetric, positive definite
    %% Output - R upper triangular matrix such that R*R' = A
    %%
    n = size(A, 1); % Obtain number of rows or columns (quadratic!)
    R = zeros(n); % Upper triangular matrix

    R(1,1) = sqrt(A(1,1)) % Initialize first value

    for j=2:n
        for i=1:j-1
            sum1 = 0
            for k=1:i-1
                sum1 = sum1 + R(k,i)*R(k,j);
            end
            R(i,j)=(A(i,j)-sum1)/R(i,i);
        end
        sum2 = 0;
        for k=1:j-1
            sum2 = sum2 + R(k,j)*R(k,j);
        end
        R(j,j)=sqrt(A(j,j)-sum2);
    end
end
```

Thomas Algorithm

Adaptation of LU algorithm to tridiagonal matrix.

L will be bidiagonal - main diagonal will be 1 - subdiagonal will be factors ß
U will be bidiagonal - main diagonal will be α and super diagonal will be factors c

```
function [L,U,x] = ThomasAlgorithm(A,b)
    %% Input:    A  quadratic tridiagonal
    %% Output:
    % L         lower matrix with one diagonal + 1 subdiag
    % U         upper matrix with main diag + upper diag of A
    % x         solution to the systems
    %%
    n=size(A,1);
    L = eye(n);
    U = zeros(n);

    U = U + triu(A,1) % Populate upper diag of U

    U(1,1) = A(1,1) % Default the first value
    for i=2:n
        L(i,i-1) = A(i,i-1) / U(i-1,i-1)
        U(i,i) = A(i-1,i-1) - L(i,i-1) * U(i-1,i)
    end

    y = L\b'
    x = U\y
end
```

Complexity:

- Forward Subst. n^2
- Backward Subst. n^2
- LU decomp: 8n

2*n^2 + n

Gram Schmidt orthonormalization

Orthonormal vectors (v,w,...)

- Their vector product is 0
- Their second norm is 1

Orthogonal Matrix if M'*M = I

Task: given a set of $n \leq d$ vectors $\{\mathbf{u}_1, \ldots, \mathbf{u}_n\} \subset \mathbb{R}^d$, find a set of n orthonormal vectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\} \subset \mathbb{R}^d$ spanning the same space.

Algorithm GS($[\mathbf{u}_1, \ldots \mathbf{u}_n]$):

$$\mathbf{e}_1 = \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|_2}$$

for $j = 2, \ldots, n$

$$\mathbf{v}_j = \mathbf{u}_j - \sum_{i=1}^{j-1} \langle \mathbf{u}_j, \mathbf{e}_i \rangle \mathbf{e}_i$$

$$\mathbf{e}_j = \frac{\mathbf{v}_j}{\|\mathbf{v}_j\|_2}$$

return $[\mathbf{e}_1, \ldots, \mathbf{e}_n]$

You are subtracting the projection part from uj when assigning it to vj

```
function [e] = GramSchmidt(A)
    %% Given the input we must find a set of orthonormal vectors spanning the same space
    %% Input - A is an array of column vectors
    %% Output - orthonormal vectors spanning the same space
    %%
    % Initializing Variables
    n = size(A,2)
    e = zeros(n)

    % Setting the first column vector
    u1 = A(:,1)
    e(:,1) = u1 / norm(u1)


    for j=2:n  % For each column
        u_j = A(:,j)

        % Calculating the Projection
        projection = 0;
        for k=1:j-1
            e_k = A(:,k);
            projection = projection + dot(u_j,e_k) * e_k;
        end

        v = u_j - projection
        e(:,j) = (v / norm(v))
    end
end
```

QR decomposition

- Can be used on rectangular matrices provided it is full rank
- Rank - maximum number of linearly independent columns of A

If A is rectangular and full rank there exists

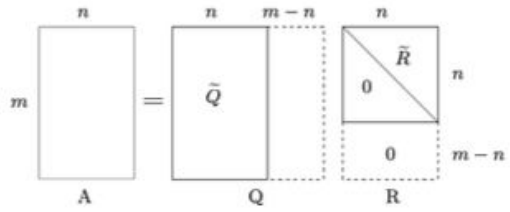
- Q of size m x m - Q' * Q = I
- R of size m x n - An upper triangular matrix and a bottom part of zeros

A = Q*R

If A is full rank the decomposition is unique (for A square it means non singular)

Modified Gram-Schmidt Orthogonalization:

FOR $j = 1, \ldots, n$
  $\mathbf{q}_j = \mathbf{a}_j$
  FOR $i = 1, \ldots, j-1$
    $r_{ij} = \langle \mathbf{q}_j, \mathbf{q}_i \rangle$ *or* $\text{proj}_{\mathbf{q}_i}(\mathbf{q}_j)$
    $\mathbf{q}_j = \mathbf{q}_j - r_{ij}\mathbf{q}_i$ $\mathbf{q}_i$ *are known*
  END
  $r_{jj} = \|\mathbf{q}_j\|$
  $\mathbf{q}_j = \mathbf{q}_j / r_{jj}$ *normalize* $\mathbf{q}_j$
END



Very stable with respect to round off

QR Factorization for a square matrix

- Q - is output of Gram-Schmidt on A
- E - is the projection of A on Q

```
function [Q,R] = QRFactorization(A)
    %% Given the input we must find a set of orthonormal vectors * a projection = A; Q*R = A
    %% Input - A is an array of column vectors
    %% Output - Q - being the Gram Schmidt orthogonal vectors, R projection of A on Q
    %%
    % Initializing Variables
    [n, m] = size(A)
    Q = zeros(n,m)
    R = zeros(n,n)

    for j=1:n
        q_j = A(:,j);
        for i =1:j-1
            R(i,j) = Q(:,i)' * A(:,j)
            q_j = q_j - R(i,j) * Q(:,i); % subtracting the projection
        end   % v is now perpendicular to all q1  - qj-1
        R(j,j) = norm(q_j)
        Q(:,j) = q_j / R(j,j);  % Normalizing v to be the next unit vector
    end
end
```

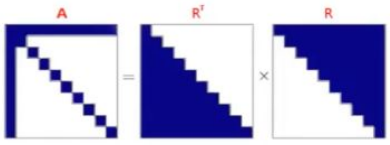
The QR factorization is the standard approach to solve the normal equations that arise in least-square problems, because it is more robust and stable with respect to round-off errors, but more expensive if m ≫ n.

7 - Iterative Methods

We've neglected the cost of data movement in direct methods which is quite high

Fill in - when we overwrite an original zero by a nonzero element

- Happens when we decomposition sparse matrices for an example
- We could reorder row and columns but not in course scope



We can avoid fill in by using iterative methods

Ax = b
- A is non singular (there is a unique solution)

We build a sequence of iterators k that converge to the exact solution x if limes goes to infinity

Iterative method:

$\{\mathbf{x}^{(k)},\ k \geq 0\}$ that **converges** to $\mathbf{x}$,

$$\lim_{k \to \infty} \mathbf{x}^{(k)} = \mathbf{x}$$

Advantages

- A is large but sparse
- Rough approx. of x is enough
- We need a good first guess for x though
- No need to store the matrix A

Underlying Idea: Compute the product Av for any vector v is relatively inexpensive

General form of iterative Methods based on splitting

- We split Matrix A into M & N such that A = M - N
- M**x** = N**x** + b
  - X on the left as new iterate
  - X on the right as old iterate
- M**x(k+1)** = N**x(k)** + **b**
  - Phi(x) = $M^{-1}N\mathbf{x}^{(k)} + M^{-1}\mathbf{b}$
  - B = $M^{-1}N$
  - g = M^{-1}
- **1st Normal Form**: x(k+1) = Bx(k) + g
  - We invert M to have the recursive definition: $\mathbf{x}^{(k+1)} = M^{-1}N\mathbf{x}^{(k)} + M^{-1}\mathbf{b}$

Consistency relation:
$\mathbf{x} = B\mathbf{x} + \mathbf{g}$

$B = (I - M^{-1}A)$: iteration matrix
$\mathbf{g} = (I - B)A^{-1}\mathbf{b}$.

(Important for algorithm properties)

  - It provides **Consistency Relation**
  - Never used for implementation!
- **2nd Normal Form**

Second normal form:
$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + M^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$

(Important for implementation)

-

Foundation of iterative methods

Error and convergence

Spectral radius:
$\rho(B) = \max\{|\lambda_1|, |\lambda_2|, \ldots, |\lambda_n|\}$

Error at iteration k: $\mathbf{e}^{(k)} = \mathbf{x} - \mathbf{x}^{(k)}$

$\mathbf{e}^{(k+1)} = B\mathbf{e}^{(k)}$

B is s.p.d.:
$\|\mathbf{e}^{(k+1)}\| = \|B\mathbf{e}^{(k)}\| \leq \lambda_{max}\|\mathbf{e}^{(k)}\|$

More properly:
$\|\mathbf{e}^{(k+1)}\| \leq \rho(B)\|\mathbf{e}^{(k)}\|$

$$\begin{aligned}\|\mathbf{e}^{(k)}\| &\leq \rho(B) \quad \|\mathbf{e}^{(k-1)}\| \\ &\leq (\rho(B))^2\|\mathbf{e}^{(k-2)}\| \\ &\leq \ldots\end{aligned}$$

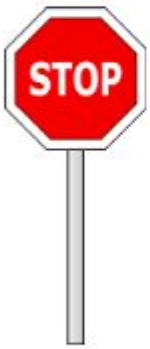
$$\boxed{\|\mathbf{e}^{(k)}\| \leq (\rho(B))^k\|\mathbf{e}^{(0)}\|}$$

Convergence:
The iterative method $\mathbf{x}^{(k+1)} = B\mathbf{x}^{(k)} + \mathbf{g}, \quad k \geq 0$, with B satisfying the consistency relation, **converges** $\forall \mathbf{x}^{(0)}$ **iif** $\rho(B) < 1$.
The smaller $\rho(B)$ the faster is the convergence.

Remark: if you know $\rho(B)$, you can estimate $k_{\min}$ to damp $\|\mathbf{e}^{(0)}\|$ by a given factor.

Foundation of iterative methods

Stopping criteria



Test on the residual:

$\|\mathbf{r}^{(k_{\min})}\| \leq \epsilon \|\mathbf{b}\|$

so: $\frac{\|\mathbf{e}^{(k_{\min})}\|}{\|\mathbf{x}\|} \leq K(A) \frac{\|\mathbf{r}^{(k_{\min})}\|}{\|\mathbf{b}\|} = \epsilon \mathcal{K}(\mathcal{A})$

Test on the increment:

$\|\delta^{(k_{\min})}\| = \|x^{(k_{\min}+1)} - x^{(k_{\min})}\| \leq \epsilon$

For B s.p.d., $\|\mathbf{e}^{(k)}\| \leq \frac{1}{1-\rho(B)} \|\delta^{(k_{\min})}\|$

Test on the relative increment:

$\frac{\|\delta^{(k_{\min})}\|}{\|\mathbf{b}\|} \leq \epsilon$

For B s.p.d., $\frac{\|\mathbf{e}^{(k)}\|}{\|\mathbf{b}\|} \leq \frac{1}{1-\rho(B)} \epsilon$

General Form of relaxation Methods

Residual at iteration k: $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$

Second normal form: $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + M^{-1}\mathbf{r}^{(k)}$

We can rewrite the second normal form

$$\boxed{M(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = \mathbf{r}^{(k)}}$$

We change the notation to a more general definition of iterative methods

- Only one Matrix P - Preconditioning Matrices
- P is nonsingular
- Usually easier structure than A

M(x(k+1) - x(k)) = r(k)

P(x(k+1) -x(k)) = alpha(k) r(k)

Pz = r(k) Then compute x(k+1) = x(k) + alpha(k) + z(k)

If alpha(k) is constant = stationary methods Else = dynamic methods

Jacobi Method

We use the Diagonal of A as our P Preconditioning Matrix

P = Diag(A)
alpha = 1

D x(k+1) - x(k) = r(k)

We need to solve **D z(k) = r(k)**

Key thing is that the linear system we're solving is much smaller than usual because it's just the diagonal.

If A ∈ Rn×n is strictly diagonally dominant by row, then the Jacobi method converges.

```
function [x] = JacobiMethod(A,b,x0,tol,itMax)
    %% Input
    % A     square matrix no zeros in diag
    % b     Solution to Ax
    % x0    Start guess
    % tol   error tolerance
    % itMax max iteration
    %% Output - x Approximated result for Ax = b
    P = diag(diag(A)); % Building D
    x = x0;
    r = b - A*x;
    rel_error = tol * 2; % norm(x - x0)/norm(x);
    it = 1;

    while(rel_error > tol & it < itMax)
        x_prev = x;

        z = P\r; % Solve Dz = r

        x = x + z;
        r = b - A*x; % calculating new residual
        rel_error = norm(x - x_prev)/norm(x); % error
        it = it + 1;
    end
end
```

Gauss-Seidel Method

Gauss-Seidel method:
$P = D - E, \qquad \alpha_k = \alpha = 1$

- $D = diag(a_{11}, a_{22}, \ldots, a_{nn})$
- $E =$ negative triangular part of A below main diagonal:

$$e_{ij} = \begin{cases} -a_{ij} & \text{if } j < i \\ 0 & \text{elsewhere} \end{cases} \qquad \text{for } i = 2, \ldots, n$$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + (D - E)^{-1}\mathbf{r}^{(k)}, \qquad k \geq 0$$

The same as Jacobi but a different P Matrix is used

Jacobi Vs Guass-Seidel convergence:
Let $A \in \mathbb{R}^{n \times n}$ be tridiagional, non-singular, with $a_{ii} \neq 0 \ \forall i$.
Then, Jacobi and Guass-Seidel methods are **either both** divergent or both convergent.
In the latter case, Guass-Seidel is faster, and $\rho(B_{\mathrm{GS}}) = \rho(B_{\mathrm{J}})^2$.

```
function [x] = Gauss_SeidelMethod(A,b,x0,tol,itMax)
   %% Input
   % A     square matrix no zeros in diag
   % b     Solution to Ax
   % x0    Start guess
   % tol   error tolerance
   % itMax max iteration
   %% Output - x Approximated result for Ax = b
   %%
   P = tril(A) % Building D
   x = x0;
   r = b - A*x;
   rel_error = tol * 2; % norm(x - x0)/norm(x);
   it = 1;

   while(rel_error > tol & it < itMax)
      x_prev = x;

      z = P\r; % Solve Dz = r

      x = x + z;
      r = b - A*x; % calculating new residual
      rel_error = norm(x - x_prev)/norm(x); % error
      it = it + 1;
   end
end
```

Successive over-relaxation

ω-versions: $\mathbf{x}^{(k+1)} = \omega \mathbf{x}^{(k+1)} + (1-\omega)\mathbf{x}^{(k)}$

In P, we use the modified $D_\omega = \frac{1}{\omega} D$:

- **under-relaxation**: $0 < \omega < 1$,
- **over-relaxation**: $1 < \omega < 2$

In particular, over-relaxation Gauss-Seidel method

Successive over-relaxation (SOR): $P = \frac{1}{\omega} D - E$ $\quad 1 < \omega < 2$

$$x_i^{(k+1)} = (1-\omega) x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \qquad i = 1, \ldots, n$$

It is faster than GS, but how to choose ω?

- try a few iterations with various values of ω
- choose the most promising one

- Use under relaxation when you have a problem with oscillations
  - Make non convergent systems convergent
- Over relaxation applied to a system that is already convergent
  - Will make it generally faster

```
function [x] = SuccessiveOverRelaxation(A,b,w,x0,tol,itMax)
    %% Input
    % A     square matrix no zeros in diag
    % b     Solution to Ax
    % x0    Start guess
    % tol   error tolerance
    % itMax max iteration
    %% Output - x Approximated result for Ax = b
    %%
    P = (1/w) .* diag(diag(A)) ; % Building D
    LowerTriWithoutDiag = tril(A) - diag(diag(A));
    P = P + LowerTriWithoutDiag;
    x = x0;
    r = b - A*x;
    rel_error = tol * 2; % norm(x - x0)/norm(x);
    it = 1;

    while(rel_error > tol & it < itMax)
        x_prev = x;

        z = P\r; % Solve Dz = r

        x = x + z;
        r = b - A*x; % calculating new residual
        rel_error = norm(x - x_prev)/norm(x); % error
        it = it + 1;
    end
end
```

Dynamic Methods

We change alpha, not a constant 1 anymore!

- Alpha changes at every iteration

We only focus on the Non-preconditioned Matrix - When P = I

General Setup

- Constant Matrix P
- Changing alpha

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \underbrace{P^{-1}\mathbf{r}^{(k)}}_{\mathbf{z}^{(k)}}$$

$$\longrightarrow \quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$$

Algorithm:

Given $\mathbf{x}^{(0)}$, set $\mathbf{r}^{(0)}$

For $k = 0, 1, \ldots$

compute the **search direction** $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$

compute the **step size** α_k

update the solution $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$

update the residual $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k A\mathbf{z}^{(k)}$

Gradient Descent Method

P = I

Z(k) = r(k)

We just need to compute alpha

General definition of step length:

$$\alpha_k = \frac{\left(\mathbf{z}^{(k)}\right)^{\mathrm{T}} \mathbf{r}^{(k)}}{\left(\mathbf{z}^{(k)}\right)^{\mathrm{T}} A\mathbf{z}^{(k)}}$$

Preconditioned gradient method:

generic P

Algorithm:

Given $\mathbf{x}^{(0)}$, set $\mathbf{r}^{(0)}$

For $k = 0, 1, \ldots$

solve $P\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$

compute $\mathbf{s}^{(k)} = A\mathbf{z}^{(k)}$

compute $\alpha_k = \frac{\left(\mathbf{z}^{(k)}\right)^{\mathrm{T}} \mathbf{r}^{(k)}}{\left(\mathbf{z}^{(k)}\right)^{\mathrm{T}} \mathbf{s}^{(k)}}$,

update $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{z}^{(k)}$

update $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{s}^{(k)}$

The gradient method:

$P = I$

Algorithm:

Given $\mathbf{x}^{(0)}$, set $\mathbf{r}^{(0)}$

For $k = 0, 1, \ldots$

compute $\mathbf{s}^{(k)} = A\mathbf{r}^{(k)}$

compute $\alpha_k = \frac{\left(\mathbf{r}^{(k)}\right)^{\mathrm{T}} \mathbf{r}^{(k)}}{\left(\mathbf{r}^{(k)}\right)^{\mathrm{T}} \mathbf{s}^{(k)}}$,

update $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{r}^{(k)}$

update $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \alpha_k \mathbf{s}^{(k)}$

Convergence:

If A, P, are s.p.d., the preconditioned gradient method **converges**.

```
function [x] = GradientMethod(A,b,x0,tol,itMax)
    %% Input
    % A     square matrix no zeros in diag
    % b     Solution to Ax
    % x0    Start guess
    % tol   error tolerance
    % itMax max iteration
    %% Output - x Approximated result for Ax = b
    %%
    x = x0;
    r = b - A*x;
    rel_error = tol * 2; % norm(x - x0)/norm(x);
    it = 1;

    while(rel_error > tol & it < itMax)
        x_prev = x;

        s = A*r;
        alpha = ((r')*r) / ((r')*s)

        x = x + alpha*r;
        r = r - alpha*s; % calculating new residual
        rel_error = norm(x - x_prev)/norm(x); % error
        it = it + 1;
    end
end
```

```
function [x] = GradientMethodPreCon(A,b,x0,tol,itMax)
    %% Input
    % A     square matrix no zeros in diag
    % b     Solution to Ax
    % x0    Start guess
    % tol   error tolerance
    % itMax max iteration
    %% Output - x Approximated result for Ax = b
    %%
    P = eye(size(A,1)); % Precondition Matrix diag used could be something else
    x = x0;
    r = b - A*x;
    rel_error = tol * 2; % norm(x - x0)/norm(x);
    it = 1;

    while(rel_error > tol & it < itMax)
        x_prev = x;

        z = P\r; % Solve Dz = r
        s = A*z;
        alpha = ((z')*r) / ((z')*s)

        x = x + alpha*z;
        r = r - alpha*s; % calculating new residual
        rel_error = norm(x - x_prev)/norm(x); % error
        it = it + 1;
    end
end
```

If A, P, are s.p.d., the preconditioned gradient method converges.

8 - Eigenvalue Problems

Eigenvalue problem

Problem definition

Eigenvalue problem:
Given $A \in \mathbb{R}^{n\times n}$,
find λ and $\mathbf{x} \neq \mathbf{0}$ so that

$$A\mathbf{x} = \lambda\mathbf{x}$$

or $(\lambda I - A)\mathbf{x} = \mathbf{0}$

- λ eigenvalue, scalar
- $\mathbf{x}$ eigenvector, with n components
- $\Lambda(A)=\{\lambda_1, \dots, \lambda_n\}$: spectrum

λ eigenvalue iff:

- **homogeneous linear system**: $(\lambda I - A)\,\mathbf{x} = \mathbf{0}$
- columns (or rows) of $(\lambda I - A)$ linearly dependent
- $\det(\lambda I - A) = 0$.

Eigenvectors:

- if $\mathbf{x}$ is eigenvector, $\alpha\mathbf{x}$ is too ($\alpha \neq 0$)
- often, $\|\mathbf{x}\|_2 = 1$
- right eigenvector $\mathbf{x} \neq \mathbf{0}$: $A\mathbf{x} = \lambda\mathbf{x}$
- (left eigenvector $\mathbf{w} \neq \mathbf{0}$: $\mathbf{w}^{\mathrm{T}}A = \lambda\mathbf{w}^{\mathrm{T}}$)

The eigenpairs $(\lambda_j, \mathbf{x}_j)$ characterizes the "behavior" of A.

Given $\mathbf{y} \in \mathbb{R}^n$ and n eigenpairs with $\{\mathbf{x}_j\}$ **linearly independent**.

- $\mathbf{y} = \sum_{j=1}^{n} \alpha_j \mathbf{x}_j$
- $A\mathbf{y} = \sum_{j=1}^{n} \alpha_j A\mathbf{x}_j = \sum_{j=1}^{n} \alpha_j \lambda_j \mathbf{x}_j$

The characteristic polynomial

A 2×2 example

Task: Compute the eigenvalues of $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in \mathbb{R}^{2\times 2}$ by solving $\det(\lambda I - A) = 0$.

$$\det(\lambda I - A) = \det \begin{bmatrix} \lambda - a_{11} & -a_{12} \\ -a_{21} & \lambda - a_{22} \end{bmatrix} = 0$$

$$\rightarrow (\lambda - a_{11})(\lambda - a_{22}) - a_{12}a_{21} = \lambda^2 - (a_{11} + a_{22})\lambda + (a_{11}a_{22} - a_{12}a_{21}) = 0$$

Remind: second-order equation in the form $a\lambda^2 + b\lambda + c = 0$
solution is based on the **discriminant** $\Delta = b^2 - 4ac$: $\lambda_{1,2} = \dfrac{-b \pm \sqrt{\Delta}}{2a}$

- $\Delta > 0$, two real solutions $\lambda_1 \neq \lambda_2$
- $\Delta = 0$, two coincident real solution $\lambda_1 = \lambda_2$
- $\Delta < 0$, two complex conjugate solution $\lambda_1 = \overline{\lambda_2}$

a) $\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$ $\quad \Delta = 5^2 - 4(4 - 2\cdot 2) = 25$, $\lambda_1 = 5,\ \lambda_2 = 0.$

b) $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ $\quad \Delta = 0 - 4(0 + 1) = -4$, $\lambda_1 = -i,\ \lambda_2 = i.$

Eigenvalues as roots of the **characteristic polynomial** $p_A(\lambda) \in \mathbb{P}^n$:

$$\boxed{p_A(\lambda) = \det(\lambda I - A) = (\lambda - \lambda_1)(\lambda - \lambda_2)\dots(\lambda - \lambda_n) = 0}$$

Algebraic multiplicity: number of roots of p_A that have the same value.

- $A \in \mathbb{R}^{n\times n}$: p_A has n real or complex roots, but not necessarily distinct
  e.g. if k roots equal to each other and their value is λ_j, then λ_j is an eigenvalue with algebraic multiplicity k
- **simple eigenvalues** if algebraic multiplicity is 1.

If λ_j has $k > 1$, what about the associated eigenvectors?
Geometric multiplicity: number of linearly independent eigenvectors associated to λ_j.

Defective matrix not a complete, linearly independent, set of eigenvector:

- at least one eigenvalues with geometric multiplicity $\leq$ algebraic multiplicity;
- A is **not diagonalizable**.

Power Method

The idea

Assumption: Repeated multiplication of a random vector $\mathbf{y}^{(k)}$ by a matrix A yields vectors that eventually tend towards the direction of the dominant eigenvector.

given $\mathbf{x}^{(0)} \in \mathbb{C}^n$
$\mathbf{y}^{(0)} = \mathbf{x}^{(0)} / \|\mathbf{x}^{(0)}\|$

for $k = 1, 2, \ldots$

$$\mathbf{x}^{(k)} = A\mathbf{y}^{(k-1)}$$

$$\mathbf{y}^{(k)} = \frac{\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|}$$

Hypotheses:

- $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$,
- $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ are linearly independent,
- $\mathbf{x}^{(0)}$ has a component in direction $\mathbf{x}_1$ (i.e, $\alpha_1 \neq 0$).

At $k \geq 1$: $$\mathbf{y}^{(k)} = \beta^{(k)} \sum_{i=1}^{n} \alpha_i \lambda_i^k \mathbf{x}_i = \lambda_1^k \beta^{(k)} \left(\alpha_1 \mathbf{x}_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1} \right)^k \mathbf{x}_i \right)$$

Under the hypotheses: *(proof in the notes, if interested)*

$$\lim_{k\to\infty} \left(\frac{\lambda_i}{\lambda_1} \right)^k = 0, \quad \text{and} \quad \lim_{k\to\infty} |\lambda_1^k \beta^{(k)} \alpha_1| = 1$$

$$\boxed{\lim_{k\to\infty} \mathbf{y}^{(k)} = \mathbf{x}_1}$$

Observation: if $(\lambda, \mathbf{x})$ eigen-pair, $$\frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{\mathbf{x}^T \lambda \mathbf{x}}{\mathbf{x}^T \mathbf{x}} = \frac{\lambda \|\mathbf{x}\|_2^2}{\|\mathbf{x}\|_2^2} = \lambda$$

```
function [eigenvector,eigenvalue] = PowerMethod(A,x0,tol,itMax)
    %% Input
    %   A    -   a square matrix
    %   x    -   n initial vector
    %   tol  -   error tolerance
    %   itMax - max Iterations
    %% Output
    %   eigenvector - dominant eigenvector
    %   eigenvalue  - dominant eigenvalue
    %%
    n = size(A,1);
    x = zeros(n,itMax)
    y = zeros(n,itMax)
    lambda = zeros(1,itMax);


    eigenvector = zeros(1,n)
    eigenvalue = 0

    x(:,1) = x0; % initial vecotr
    y(:,1) = x(:,1) / norm(x(:,1));


    k = 2; %Iterator
    while(k<itMax & norm(y(:,k)-y(:,k-1)) > tol)
        x(:,k) = A * y(:,k-1);
        y(:,k) = x(:,k) / norm(x(:,k));
        lambda(k) = (y(:,k)') * (A*y(:,k))

        k = k+1
    end
    eigenvector = y(:,k-1)
    eigenvalue = lambda(k-1)
end
```

Inverse Power Method

We want the smallest not the biggest eigenvalue in modulus

- Therefore we just invert the Matrix A
- We need to solve a system at each iteration - quite intensive

```
function [eigenvector,eigenvalue] = InversePowerMethod(A,x0,tol,itMax)
    %% Input
    %   A   -   a square matrix
    %   x   -   n initial vector
    %   tol -   error tolerance
    %   itMax - max Iterations
    %% Output
    %   eigenvector - dominant eigenvector
    %   eigenvalue  - dominant eigenvalue
    %%
    n = size(A,1);
    x = zeros(n,itMax)
    y = zeros(n,itMax)
    lambda = zeros(1,itMax);


    eigenvector = zeros(1,n)
    eigenvalue = 0

    x(:,1) = x0; % initial vector
    y(:,1) = x(:,1) / norm(x(:,1));


    k = 2; %Iterator
    while(k<itMax & norm(y(:,k)-y(:,k-1)) > tol)
        x(:,k) = A \ y(:,k-1);
        y(:,k) = x(:,k) / norm(x(:,k));
        lambda(k) = (y(:,k)') * x(:,k);

        k = k+1;
    end
    eigenvector = y(:,k-1)
    eigenvalue = 1/lambda(k-1)
end
```

Inverse Power Method with Shift

- Used when we're not looking for the biggest or smallest value but something in between
- We use a shifting constant alpha
- Whichever eigenvalue is the closest to alpha will be the new dominant eigenvalue

A * **v** = **w** * **v**

$$\text{Eigenvec.} \quad A \cdot \vec{v} = \lambda \vec{v} \qquad | - \alpha I \vec{v}$$

$$(A - \alpha I)\vec{v} = (\lambda - \alpha)\vec{v}$$

$$\frac{\vec{v}}{(\lambda - \alpha)} = (A - \alpha I)^{-1} \cdot \vec{v}$$

$$\underbrace{(A - \alpha I)^{-1}}_{\text{Eigenvec.}} \cdot \vec{v} = \frac{1}{(\lambda - \alpha)} \cdot \vec{v}$$

But the converge is better than the others.

```
function [eigenvector,eigenvalue] = InversePowerMethodShift(A,x0,alpha,
tol,itMax)
    %% Input
    %   A   -   a square matrix
    %   x   -   n initial vector
    %   alpha - value to find the closest eigenvalue to
    %   tol -   error tolerance
    %   itMax - max Iterations
    %% Output
    %   eigenvector - dominant eigenvector
    %   eigenvalue  - dominant eigenvalue
    %%
    n = size(A,1);
    x = zeros(n,itMax)
    y = zeros(n,itMax)
    lambda = zeros(1,itMax); % Eigenvalues belonging to B

    x(:,1) = x0; % initial vector
    y(:,1) = x(:,1) / norm(x(:,1));

    B = A - (alpha * eye(n));
    k = 2; %Iterator
    while(k<itMax & norm(y(:,k)-y(:,k-1)) > tol)

        x(:,k) = B \ y(:,k-1);
        y(:,k) = x(:,k) / norm(x(:,k));

        lambda(k) = (y(:,k)') * x(:,k)

        k = k+1
    end

    eigenvector = y(:,k-1)
    eigenvalue = 1/lambda(k-1)+alpha
end
```

Inverse Power Method with dynamic shift

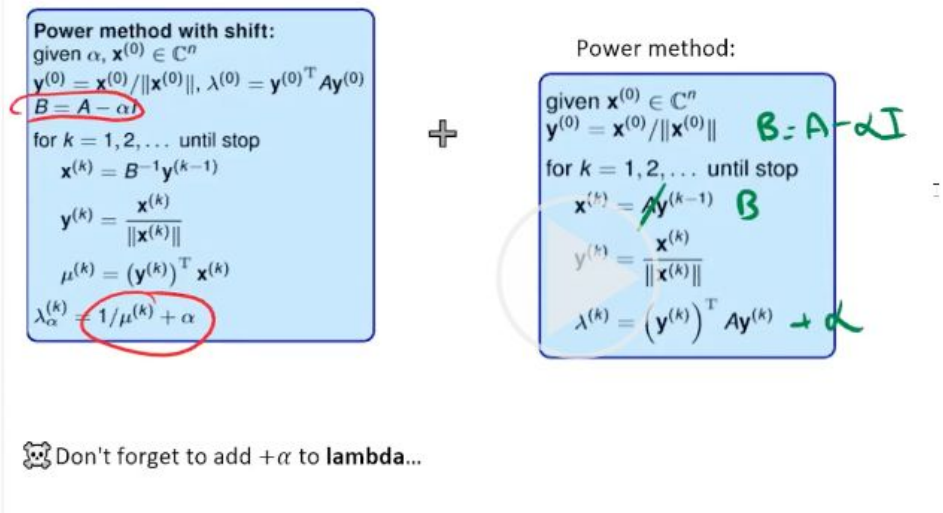
Changing the alpha by which we multiple get B at each iteration
We can't perform the factorization of B outside of the loop. - Bad

```
function [eigenvector,eigenvalue] = InversePowerMethodShiftDynamic(A,x0,alpha, tol,itMax)
    %% Input
    %   A   -   a square matrix
    %   x   -   n initial vector
    %   alpha - value to find the closest eigenvalue to
    %   tol -   error tolerance
    %   itMax - max Iterations
    %% Output
    %   eigenvector - dominant eigenvector
    %   eigenvalue  - dominant eigenvalue
    %%
    n = size(A,1);
    x = zeros(n,itMax)
    y = zeros(n,itMax)
    lambda = zeros(1,itMax); % Eigenvalues belonging to B
    lambdaAlpha = zeros(1,itMax); % Eigenvalues belonging to A


    x(:,1) = x0; % initial vector
    y(:,1) = x(:,1) / norm(x(:,1));
    lambdaAlpha(1) = (y(:,1)') * (A * y(:,1));


    k = 2; %Iterator
    while(k<itMax & norm(y(:,k)-y(:,k-1)) > tol)

        B = A - (lambdaAlpha(k-1) * eye(n));

        x(:,k) = B \ y(:,k-1);
        y(:,k) = x(:,k) / norm(x(:,k));

        lambda(k) = (y(:,k)') * x(:,k);
        lambdaAlpha(k) = 1 / (lambda(k)) + alpha;

        k = k+1;
    end

    eigenvector = y(:,k-1)
    eigenvalue = lambdaAlpha(k-1)
end
```

Power Method with Shift

Implement the thing on the right side
Give a shift value to makes sure we calculate the biggest offset so we can get the **smallest eigenvalue without any system inversion (efficient)**

We calculate the max eigenvalue = tau and we shift by a to find the furthest eigenvalue away from tau




```
function [lambda, xx, iter, lambdas, xxs] = powershift(A, alpha, tol, itMax, x0)
    % Powershift method to compute largest eigenvalue of a matrix A-alpha*I
    %% Initalisation
    n = size(A,1);
    B=A-eye(n)*alpha;
    err = 1e10;
    iter = 0;
    xx = x0;
    yy = xx/norm(xx,2);

    lambdas = zeros(1,itMax);
    xxs=zeros(n,itMax);


    while err>= tol && iter<itMax

        % Calculate the new eigenvector
        xx = B*yy;
        yy = xx/norm(xx,2);

        % Calculate the new eigenvalue
        lambda = yy'*(B*yy);

        % New iter and storage of the history
        iter = iter + 1;
        xxs(:,iter) = yy; % Normalised output for eigenvector estimation

        lambdas(iter) = lambda + alpha; % Don't forget the shift

        if iter>1
            err = abs(lambdas(iter) - lambdas(iter-1)) / abs(lambdas(iter) - alpha);
        end
    end

    % Just sending the truncated important data back
    lambda = lambda + alpha;
    lambdas = lambdas(1:iter);
    xxs = xxs(:,1:iter);
    xx = yy; % Final normalized xx
end
```

9 - Numerical Differentiation & Integration

Numerical differentiation on equi-distributed nodes

Numerical Integration on equi-distributed nodes

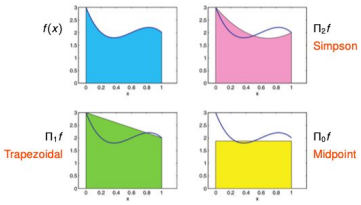
- The primitive function is not known or too complex for us to find out

We look for an approximation of

$$\boxed{I(f) = \int_a^b f(x)\mathrm{d}x}$$

Newton-Cotes formula

- Very common
- Formula to integrate over equally spaced nodes
- They replace the function to be integrated (or the given tabulated data) by a (piece-wise) polynomial interpolation, which is easy to integrate.
- There are different flavours of formulas used



We want to find a polynomial of degree n for this

$$\boxed{I(f) = \int_a^b f(x)\mathrm{d}x \approx \int_a^b \Pi_n f(x)\mathrm{d}x}$$

	Error composite $\max\lvert I(f) - I^C(f)\rvert$	Degree Exactness	Simple $\max\lvert I(f) - I(f)\rvert$
Midpoint	If $f \in C^2([a,b])$, $\frac{b-a}{24} H^2 \max\lvert f''(\xi)\rvert$	1	$\frac{(b-a)^3}{24} \max\lvert f''(\xi)\rvert$
Trapezoidal	If $f \in C^2([a,b])$, $\frac{b-a}{12} H^2 \max\lvert f''(\xi)\rvert$	1	$\frac{(b-a)^3}{12} \max\lvert f''(\xi)\rvert$
Simpson	If $f \in C^4([a,b])$, $\frac{b-a}{180} \frac{H^4}{16} \max\lvert f^{(4)}(\xi)\rvert$	3	$\frac{(b-a)^5}{180 \cdot 16} \max\lvert f^{(4)}(\xi)\rvert$

max is for an $\xi \in [a,b]$.

Degree of exactness
is the maximum integer $r \geq 0$ for which the approximate integral (produced by the quadrature formula) of any polynomial of degree r is equal to the **exact** integral.

- Composite formulas can drive error $\to 0$ as $h \to 0$, but round-off errors.
- **Cost**: how many f evaluations do we need to reach a certain accuracy.
- Generally, if $f(x)$ is sufficiently smooth, high order methods are more efficient.

Composite quadrature Formulas

For functions in one variable the integral from a - b is **the area under the curve**
We can do this in a **cumulative way**

- We split the interval a-b in M subintervals
- And then sum up the M areas

$$\int_a^b f(t)\mathrm{d}t = \int_a^{x_1} f(t)\mathrm{d}t + \int_{x_1}^{x_2} f(t)\mathrm{d}t + \cdots + \int_{x_{M-1}}^{b} f(t)\mathrm{d}t$$

- The degree of exactness of an n + 1-point Newton-Cotes rule is at least n, but there are some lucky cases when it is greater than n.

Midpoint Formula

Midpoint rule: $\bar{x} = \frac{a+b}{2}$

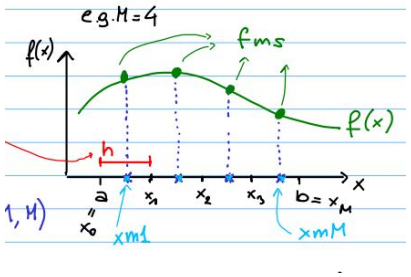
$$I_{\mathrm{mp}}(f) = (b-a)f(\bar{x})$$

Newton-Cotes Form

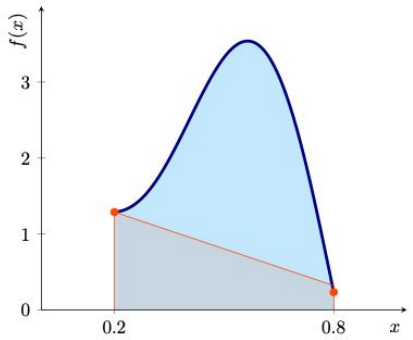
- Just take the middle and do this calculation

Composite Formula Form

```
function [AproxIntegral] = MidpointCompositeFormula(fun,a,b,M)
    %% Input
    %   fun     function in vector form .* etc.
    %   a       start interval
    %   b       end interval
    %   M       number of subintervals to create
    %% Output
    %   AproxIntegral - Area of summed subintervals
    %% Code
    h = (b-a)/M % length of an inteval

    xMidpoint1 = a + h/2;
    xMidpointM = b - h/2;

    EquiSpacedNodes = linspace(xMidpoint1, xMidpointM, M);
    funEvaluatedAtNods = fun(EquiSpacedNodes);
    AproxIntegral =  sum(funEvaluatedAtNodes) * h;
end
```



Trapezoidal formula



Newton-Cotes Form

- Just create a trapezoid from a to b

Trapezoidal rule:

$$I_{\mathrm{t}}(f) = (b-a)\frac{f(a)+f(b)}{2}$$

Composite trapezoidal formula:

$$I_{\mathrm{t}}^{\mathrm{c}}(f) = \frac{H}{2}\left[f(x_0) + 2\sum_{k=1}^{M-1} f(x_k) + f(x_M)\right]$$

Composite Formula Form

Just apply the trapezoidal formula repeatedly

In Matlab: *Trapz, cumtrapz*

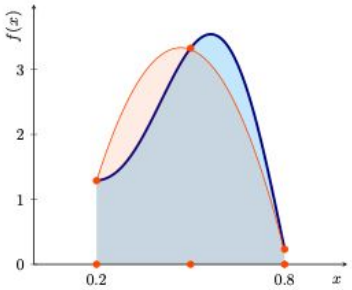
$$I_{\mathrm{t}}^{\mathrm{c}}(f) = H\frac{f(x_0)+f(x_1)}{2} + H\frac{f(x_1)+f(x_2)}{2} + \cdots + H\frac{f(x_{M-1})+f(x_M)}{2}$$

```
function [AproxIntegral] = TrapezoidalCompositeFormula(fun,a,b,M)
    %% Input
    %   fun     function in vector form .* etc.
    %   a       start interval
    %   b       end interval
    %   M       number of subintervals to create
    %% Output
    %   AproxIntegral - Area of summed subintervals
    %% Code
    h = (b-a)/M % length of an inteval

    xMidpoint1 = a + h/2;
    xMidpointM = b - h/2;

    EquiSpacedNodes = linspace(xMidpoint1 + h, xMidpointM - h, M-2); % We want interval without x1 and xM
    funEvaluatedAtNodes = fun(EquiSpacedNodes);

    AproxIntegral = h/2 * (fun(xMidpoint1) + 2 * sum(funEvaluatedAtNodes) + fun(xMidpointM));
end
```

Simpson Formula

Newton-Cotes Form

$$I_s(f) = \int_a^b \Pi_2 f \, dx \quad \text{at nodes } \{a, \bar{x}, b\}.$$



Simpson quadrature formula:

$$I_s(f) = (b-a)\,\frac{f(a) + 4f(\bar{x}) + f(b)}{6}$$

Composite Form

- Superior to Trapezoid Rule, especially when the function is smooth

For the composite formula, split the interval $[a, b]$ in M sub-intervals ($\bar{x}_k = (x_{k-1} + x_k)/2$):

$$a = x_0 \quad \bar{x}_1 \quad x_1 \quad \bar{x}_2 \quad x_2 \quad - - - - \quad x_{M-1} \quad \bar{x}_M \quad x_M = b$$

$$I_s^c(f) = H\frac{f(x_0) + 4f(\bar{x}_1) + f(x_1)}{6} + H\frac{f(x_1) + 4f(\bar{x}_2) + f(x_2)}{6} + \cdots + H\frac{f(x_{M-1}) + 4f(\bar{x}_M) + f(x_M)}{6}$$

Composite Simpson quadrature formula:

$$I_s^c(f) = \frac{H}{6}\sum_{k=1}^{M}\left[f(x_{k-1}) + 4f(\bar{x}_k) + f(x_k)\right]$$

```
function [AproxIntegral] = SimpsonComposite(fun,a,b,M)
    %% Input
    %   fun     function in vector form .* etc.
    %   a       start interval
    %   b       end interval
    %   M       number of subintervals to create
    %% Output
    %   AproxIntegral - Area of summed subintervals
    %% Code
    h = (b-a)/M % length of an inteval

    x1 = a;
    xM = b;

    EquiSpacedNodes = linspace(x1 ,xM, M);
    EquiSpacedMidPoints = linspace(x1+h/2,xM-h/2,M-1);

    funAtNodes = fun(EquiSpacedNodes)
    funAtMidpoints = fun(EquiSpacedMidPoints)

    simpsonSum = 0;
    for i=1:M-1
        simpsonSum = simpsonSum + funAtNodes(i) + 4*funAtMidpoints(i) + funAtNodes(i+1);
    end

    AproxIntegral = h/6 *  simpsonSum;
end
```

Numerical Integration of known functions

The general form of quadrature

- Given n+1 nodes

We can build integrate exactly any polynomial of degree equal and less than n

$$I(f) = \int_a^b f(x)\mathrm{d}x \approx \int_a^b \sum_{j=0}^{n} \varphi_j(x) f(x_j)\mathrm{d}x = \sum_{j=0}^{n} f(x_j) \int_a^b \varphi_j(x)\mathrm{d}x = \sum_{j=0}^{n} y_j \alpha_j \,.$$

Notice that to integrate exactly a constant function, we need $\sum_{j=0}^{n} \alpha_j = b - a$.

We want to approximate a quadrature formula that has the same degree of exactness as the polynomial of degree m

underlying polynomial approximation of degree n

$$I^{APPR.} = \int_a^b \overbrace{\Pi_m f(y)}\, dy = \int_a^b \sum_{j=0}^{m} \varphi_j(y)\, f(y_j)\, dy$$

$$I^{APPR} = \sum_{j=0}^{m} \Big(f(y_j) \underbrace{\int_a^b \varphi_j(y)\, dy}_{= \alpha_j} \Big) = \sum_{j=0}^{m} \alpha_j\, f(y_j)$$

We look for a generalization of the newton-cotes formula

Gauss Quadrature Nodes

- Provide the maximum degree of freedom
- 2m+1 degree of exactness
- The nodes are defined as the zero from the Legendre polynomial - recursively defined

Legendre polynomial $L_{n+1}(x)$:

$L_0(x) = 1, \qquad L_1(x) = x,$

$$L_{k+1}(x) = \frac{2k+1}{k+1} x L_k(x) - \frac{k}{k+1} L_{k-1}(x), \qquad k = 1, 2, \ldots$$

You calculate the n+1 legendre polynomial and the zeros are the quadrature nodes

Gauss-Legendre formula using $n+1$ nodes

- $\bar{y}_j$ are the zeros of $L_{n+1}(x)$
- $\bar{\alpha}_j = \dfrac{2}{(1-\bar{y}_j^2)[L'_{n+1}(\bar{y}_j)]^2} \quad j = 0, \ldots, n\,.$
- maximum degree of exactness of $2n+1$.

How to use in practice:

1. Choose n, e.g. the number of nodes $n+1$,
2. For the selected n, retrieve the Gauss nodes $\{\bar{y}_j\}$ and the corresponding weights $\{\bar{\alpha}_j\}$,
3. Compute $\{y_j\}$ and $\{\alpha_j\}$ mapping $\{\bar{y}_j\}$ and $\{\bar{\alpha}_j\}$ from the canonical interval $[-1, 1]$ to the general interval $[a, b]$,
4. Evaluate the function f at the nodes $\{y_j\}$,
5. Compute the quadrature $I_{appr} = \sum_{j=0}^{n} \alpha_j f(y_j)$.

- I look up the Gauss Nodes and cores. Weight online
- We scale them from the canonical interval to my a b interval
- Evaluate f at the now scaled nodes
- Compute the quadrature $I_{appr} = \sum_{j=0}^{n} \alpha_j f(y_j)$.

Gauss-Legendre-Lobatto
Includes the endpoints
Only a degree of 2n -1 exactness

- $\bar{y}_0 = -1$, $\bar{y}_n = 1$, $\bar{y}_j$ are the zeros of $L_n'(x)$ for $j = 1, \ldots, n-1$.
- $\bar{\alpha}_j = \dfrac{2}{n(n+1)[L_n(\bar{y}_j)]^2}$ $j = 0, \ldots, n$
- In MATLAB: `quadl`

CODEEE

Additional Exercise (not to be submitted)

Use a 5-point Gauss quadrature rule $I_{\text{appr}} = \sum_{j=0}^{n} \alpha_j f(y_j)$ to compute

$$\int_{-3}^{3} \exp(x)\mathrm{d}x$$

(a) Compute analytically the expression of the Legendre polynomial $L_5(x)$

(b) Compute its roots in the interval $x \in [-1, 1]$, which are the quadrature nodes $\{\bar{y}_j\}$
Hint: you can plot the polynomial to identify different initial guesses to be used in an appropriate iterative method.

(c) Compute the quadrature weights $\{\bar{\alpha}_j\}$, by integrating the Lagrangian polynomials $\phi_j(x)$ or using the expression for the *Gauss-Legendre formula* (see slides).

(d) Compute the quadrature (remind the interval transformation) and compare to the exact value.

10 - Nonlinear Equations & Numerical Optimization

Systems of nonlinear equations

Remind some concepts from nonlinear equations in one variable

Nonlinear equation in 1 variable:

$$f(x) = 0$$

- bisection method,
- Newton's method, and variants
- (fixed point iterations.)

Newton's method:

- Initial guess: $x^{(0)}$
- For $k \geq 0$:
  $$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}.$$
- Stopping criterion.

System of nonlinear equations:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) & = 0 \\ f_2(x_1, x_2, \ldots, x_n) & = 0 \\ \ldots \\ f_n(x_1, x_2, \ldots, x_n) & = 0 \end{cases}$$

$$\mathbf{f} = \{f_1, \ldots, f_n\}^{\mathrm{T}}, \quad \mathbf{x} = \{x_1, \ldots, x_n\}^{\mathrm{T}}$$

- n equations in n variables.
- **No** matrix form $A\mathbf{x} = \mathbf{b}$.

Iterative methods to find $\mathbf{x}^*$, $\mathbf{f}(\mathbf{x}^*) = \mathbf{0}$

- initial guess $\mathbf{x}^{(0)}$;
- iterates $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$;
- stopping criteria;
- convergence (?) to $\mathbf{x}^*$.

How many $\mathbf{x}^*$?

- F is a vector
- X is a vector

Derivative of vector functions

Jacobian matrix: $J_f \in \mathbb{R}^{m \times n}$

$$J_f(\mathbf{x})_{ij} = \frac{\partial f_i}{\partial x_j}$$

$$J_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Taylor series:

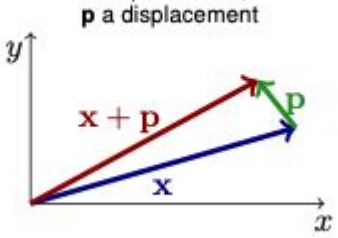
Let $\mathbf{x} = \{x_1, \ldots, x_n\}^{\mathrm{T}}$, $\mathbf{f} = \{f_1, \ldots, f_n\}^{\mathrm{T}}$, and assume that $\mathbf{f}(\mathbf{x})$ has bounded derivatives up to order at least two.

Then for a direction vector $\mathbf{p} = \{p_1, \ldots, p_n\}^{\mathrm{T}}$, the Taylor expansion for each function f_i in each coordinate x_j yields

$$\mathbf{f}(\mathbf{x} + \mathbf{p}) = \mathbf{f}(\mathbf{x}) + \mathbf{J_f}(\mathbf{x})\,\mathbf{p} + \mathcal{O}(\|\mathbf{p}\|^2)$$

Example in 2D:

$\mathbf{x}$ a point in $\mathbb{R}^2$,
$\mathbf{p}$ a displacement



Partial derivative:

$$\frac{\partial f_i(\mathbf{x})}{\partial x_j} = \lim_{h \to 0} \frac{f_i(\mathbf{x} + h\mathbf{e}_j) - f_i(\mathbf{x})}{h}$$

Note: $\mathbf{e}_j$ is jth unitary vector of $\mathbb{R}^n$.

Newton Method

From the Taylor series: $\mathbf{f}(\mathbf{x}^{(k+1)}) = \mathbf{f}(\mathbf{x}^{(k)}) + J_{\mathbf{f}}(\mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) + \mathcal{O}(\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|^2)$

We define $\mathbf{p}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ **so that:** $\mathbf{0} = \mathbf{f}(\mathbf{x}^{(k)}) + J_{\mathbf{f}}(\mathbf{x}^{(k)})\mathbf{p}^{(k)}$

- P is the direction we're moving
- Jacobian Matrix should be non singular!
    - det(J(x(k))) != 0
- In one variable we required the derivative here the jacobian matrix

Newton's method:
INPUT: $\mathbf{x}^{(0)} \in \mathbb{R}^n$
FOR $k = 1, 2, \ldots$ until stop
  solve $J_{\mathbf{f}}(\mathbf{x}^{(k)})\mathbf{p}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$
OUTPUT: $\mathbf{x}^{(k+1)}$

Stopping criterion:
on $\|\mathbf{p}^{(k)}\|$, or $\|\mathbf{f}(\mathbf{x}^{(k+1)})\|$,
relative or absolute

Solution of a **linear** system,
which must be non-singular!
$\det J_F(\vec{x}^{(k)}) \neq 0$

Convergence: If at a neighborhood of an isolated root $\mathbf{x}^*$, $J_{\mathbf{f}}(\mathbf{x})$ has bounded inverse and continuous derivatives, there exists $M = \text{const}$ s.t.

$$\|\mathbf{x}^* - \mathbf{x}^{(k+1)}\| \leq M\|\mathbf{x}^* - \mathbf{x}^{(k)}\|^2$$

provided $\|\mathbf{x}^* - \mathbf{x}^{(k)}\|$ is small enough.

in simpler words: with a "**good**" **initial guess**, it converges **quadratically**

```
x0 = [-1;1;];
tol = 10^-5;
itMax = 100;
[x, y, iter] = NewtonMethodExtendedx(@fun, @jacobian, x0, tol, itMax)

function [x, y, iter] = NewtonMethodExtended(Ffun, JacobianFun, x0, tol, itMax)
    iter = 0;
    err = tol + 1;
    x = x0; % first guess

    while (err >= tol & iter < itMax)
        J = JacobianFun(x);
        F = Ffun(x);
        delta = -J \ F;
        x = x + delta;
        err = norm(delta)
        iter = iter + 1;
    end
    y = norm(Ffun(x));
end


function F = fun(x)
    F(1,1) = x(1)^2 + x(2)^2 -1;
    F(2,1) = x(1)^2 - 2*x(1)-x(2) + 1;
end

function J = jacobian(x)
    J(1,1) = 2*x(1);
    J(1,2) = 2*x(2);
    J(2,1) = 2*x(1) -2;
    J(2,2) = -1;
end
```

Broyden Method

Idea: recursively replace $J_{\mathbf{f}}(\mathbf{x}^{(k)})$ with a suitable matrix $B^{(k)}$;
▶▶ $B(\mathbf{x}^{(0)})$ should be an *approximation* of $J_{\mathbf{f}}(\mathbf{x}^{(0)})$.

Algorithm: Broyden method:
INPUT $\mathbf{x}^{(0)} \in \mathbb{R}^n$, $B^{(0)} \in \mathbb{R}^{n\times n}$
FOR $k = 1, 2, \ldots$ until stop
  solve $B^{(k)}\mathbf{p}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$
  set $\delta\mathbf{f}^{(k)} = \mathbf{f}(\mathbf{x}^{(k+1)}) - \mathbf{f}(\mathbf{x}^{(k)})$
  compute $B^{(k+1)} = B^{(k)} + \dfrac{(\delta\mathbf{f}^{(k)} - B^{(k)}\mathbf{p}^{(k)})\mathbf{p}^{(k)\,\mathrm{T}}}{\mathbf{p}^{(k)\,\mathrm{T}}\mathbf{p}^{(k)}}$
END
OUTPUT: $\mathbf{x}^{(k+1)}$

$B^{(k)}$ is convenient approximation of $J_{\mathbf{f}}(\mathbf{x}^*)$ along $\mathbf{x}^{(k)} - \mathbf{x}^*$.

$$\lim_{k\to\infty} \frac{\|(B^{(k)} - J_{\mathbf{f}}(\mathbf{x}^*))(\mathbf{x}^{(k)} - \mathbf{x}^*)\|}{\|\mathbf{x}^{(k)} - \mathbf{x}^*\|} = 0$$

```
x0 = [-1;1;];
tol = 10^-5;
itMax = 100;
B0 = [-1.5,1.5;-3.5,-0.5]
[x, y, iter] = Broyden(@fun, B0, x0, tol, itMax)

function [x, y, k] = Broyden(Ffun, B0, x0, tol, itMax)
    err = tol + 1;
    B = B0
    B(:,:,1) = B0;
    x = x0
    x(:,:,1) = x0

    k = 1; % iterator
    while (err >= tol & k < itMax)

        F = Ffun(x(:,:,k));
        delta = -B(:,:,k) \ F;
        x(:,:,k+1) = x(:,:,k) + delta;

        F1 = Ffun(x(:,:,k+1));
        deltaF = F1 - F;

        B(:,:,k+1) = B(:,:,k) + (((deltaF-B(:,:,k)*delta)*delta')/((delta')*delta))

        err = norm(delta)
        k = k + 1;
    end
    y = norm(Ffun(x(:,:,k-1)));
end


function F = fun(x)
    F(1,1) = x(1)^2 + x(2)^2 -1;
    F(2,1) = x(1)^2 - 2*x(1)-x(2) + 1;
end
```

Numerical Optimization

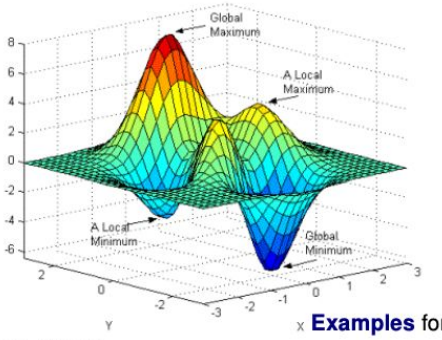
- We focus on Unconstrained Optimization
- We look for the minimum if we want the max just flip the function

Critical points for a function in n variables

- Global minimizer:
  $f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n$
- **Local minimizer:**
  $f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in B_r(\mathbf{x}^*) \subset \mathbb{R}^n$
  $B_r(\mathbf{x}^*)$ ball centered at $\mathbf{x}^*$, a region of $\mathbb{R}^n$

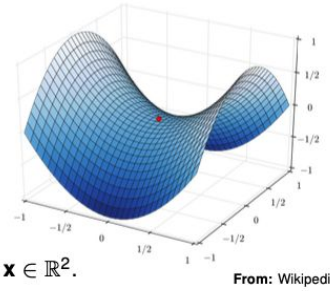
Maximizer: $f(\mathbf{x}^*) \geq f(\mathbf{x})$

Saddle point: a maximum is reached w.r.t. some variables and a minimum is reached w.r.t. other components.



From: Wikimedia

Saddle point:



From: Wikipedia

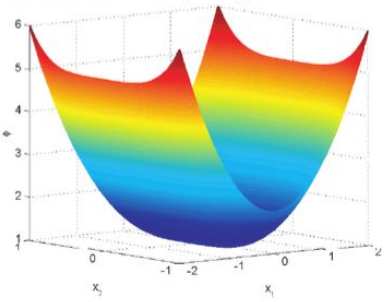
Examples for $f(\mathbf{x})$ with $\mathbf{x} \in \mathbb{R}^2$.

Some definitions for a function in n variables: derivatives

- Gradient of f at point $\mathbf{x}$:
  $$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1} \cdots \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^{\mathrm{T}}$$
- Hessian matrix of f at point $\mathbf{x}$:
  $$H_f(\mathbf{x}) = \nabla^2 f(\mathbf{x})$$
  $$h_{ij}(\mathbf{x}) = \frac{\partial^2 f}{\partial x_i \partial x_j}$$
  $$\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Note: if $f \in \mathcal{C}^2(\mathbb{R}^n)$,
then $H_f(\mathbf{x})$ is symmetric $\forall \mathbf{x} \in \mathbb{R}^n$.

Example: $f = x_1^2 + x_2^4 + 1$



From: Ascher, Greif. *A First Course in Numerical Methods*

$$\mathbf{x}^* = [0, 0]^{\mathrm{T}}$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} 2x_1 \\ 4x_2^3 \end{bmatrix}; \quad H_f(\mathbf{x}) = \begin{bmatrix} 2 & 0 \\ 0 & 12x_2^2 \end{bmatrix}$$

Conditions for having a **local minimum** at $\mathbf{x}^*$ (if $\exists r > 0 \,\mathrm{s.t.}\, f \in \mathcal{C}^2(B_r(\mathbf{x}^*))$):

- **necessary:** $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $H_f(\mathbf{x}^*)$ is positive semidefinite,
- **sufficient:** $\nabla f(\mathbf{x}^*) = \mathbf{0}$ and $H_f(\mathbf{x}^*)$ is positive definite.

Note: at saddle points: $\nabla f(\mathbf{x}^*) = \mathbf{0}$

Newton's method for minimization

- We need a the gradient to be zero and the function to be zero
- So if you use take the Jacobian Matrix of the Gradient you have the Hesse Matrix of the Original function

Optimization problem:
Find the solution $\mathbf{x}^*$ of

$$\mathbf{F}(\mathbf{x}) = \nabla \mathbf{f}(\mathbf{x}) = \mathbf{0},$$

with $J_{\mathbf{F}}(\mathbf{x}) = H_f(\mathbf{x})$.

Algorithm:
INPUT: $\mathbf{x}^{(0)} \in \mathbb{R}^n$
FOR $k = 1, 2, \ldots$ until stop
  solve $H_f(\mathbf{x}^{(k)})\mathbf{p}^{(k)} = -\nabla f(\mathbf{x}^{(k)})$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$
OUTPUT: $\mathbf{x}^{(k+1)}$

Local convergence:
If $f \in C^2(\mathbb{R}^n)$, $\nabla \mathbf{x}^* = \mathbf{0}$, $H_f(\mathbf{x}^*)$ is s.p.d, the components of $H_f(\mathbf{x})$ (i.e., $\frac{\partial^2 f}{\partial x_i \partial x_j}$) are Lipschitz continuous in a neighborhood of $\mathbf{x}^*$, then this algorithm converges **quadratically**, provided $\|\mathbf{x}^* - \mathbf{x}^{(0)}\|$ is small enough.

f is Liptschitz continuous
if $\exists L > 0$ such that:
$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|$
$\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

In other words: if f has **sufficiently regular** second order derivatives, and $\mathbf{x}^{(0)}$ is a **"good" initial guess**, the Newton's method applied to the minimization problem converges **quadratically**.

Line search (or descent) methods

We start by analyzing the taylor series

$$f(\mathbf{x} + \mathbf{p}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^{\mathrm{T}} \mathbf{p} + \mathcal{O}(\|\mathbf{p}\|^2)$$

For f(x+p) to become smaller we need the **directional derivative to be negative:**

$$\nabla f(\mathbf{x})^{\mathrm{T}} \mathbf{p} < 0.$$

We need therefore a smart way to find the best p.

Descent direction should be zero if we're already at a critical point.

Descent direction $\mathbf{d}^{(k)}$:

$$\begin{aligned} \mathbf{d}^{(k)\mathrm{T}} \nabla f(\mathbf{x}^{(k)}) &< 0 \quad && \text{if } \nabla f(\mathbf{x}^{(k)}) \neq \mathbf{0} \\ \mathbf{d}^{(k)} &= 0 && \text{if } \nabla f(\mathbf{x}^{(k)}) = \mathbf{0} \end{aligned}$$

Iterative Algorithm

- Similar algorithm to gradient of a linear system

Descent Direction

- Matrix B changing at every iteration

INPUT: $\mathbf{x}^{(0)} \in \mathbb{R}^n$, $f(\mathbf{x})$, $\nabla f(\mathbf{x})$
FOR $k = 0, 1, \ldots$ until <u>convergence</u>
  find a direction $\mathbf{d}^{(k)} \in \mathbb{R}^n$
  compute the step $\alpha_k \in \mathbb{R}$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{d}^{(k)}$
END
OUTPUT: $\mathbf{x}^{(k+1)}$

Most methods for unconstrained minimization use **descent directions** of the form

$$\mathbf{d}^{(k)} = -B^{(k)^{-1}} \nabla f(\mathbf{x}^{(k)})$$

with $B^{(k)}$ symmetric positive definite.

Some examples:

- **Gradient directions:** $\mathbf{d}^{(k)} = -\nabla f(\mathbf{x}^{(k)})$
- **Netwon's directions:**
  $\mathbf{d}^{(k)} = -\left(H(\mathbf{x}^{(k)})\right)^{-1} \nabla f(\mathbf{x}^{(k)})$ (*Newton method*)
- **Quasi Netwon's directions:**
  $\mathbf{d}^{(k)} = -\left(H_k\right)^{-1} \nabla f(\mathbf{x}^{(k)})$ e.g. *Broyden method*

- Newton Method type of line search method

How do we compute the step length alpha?

- We solve a one dimensional problem where we want to find the best step size for our function f in the direction d

Optimum $\alpha_k \in \mathbb{R}$:

- maximum variation of f along $\mathbf{d}^{(\mathbf{k})}$,
- $\mathbf{x}^{(k+1)}$ minimizer of f along $\mathbf{d}^{(\mathbf{k})}$: $\alpha_k = \operatorname{argmin}_{\alpha \in \mathbb{R}} f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(\mathbf{k})})$
- one-dimensional minimization problem

For a quadratic function:

$$\alpha_k = \frac{\mathbf{d}^{(k)\,\mathrm{T}} \mathbf{r}^{(k)}}{\mathbf{d}^{(k)\,\mathrm{T}} A \mathbf{d}^{(k)}}$$

CODE? - not really

```
tol = 0.001;
x0 = -2;
fun = @(x)  2*sin(x)+(x^2)/10;
dfun = @(x) 2*cos(x) + x/5

itMax = 1000;
[x, y, iter] = GradientDescent(fun, dfun, x0, tol, itMax)



function [x, y, iter] = GradientDescent(Ffun, grad, x0, tol, itMax)
    iter = 0;
    err = tol + 1;
    x = x0; % first guess
    stepsize = 0.01;
    while (err >= tol & iter < itMax)
        x_prev = x
        x = x - stepsize*grad(x);
        err = abs(norm(x_prev-x,2));
        iter = iter + 1;
    end
    y = norm(Ffun(x));
end
```

Non-linear least square method

- Instead of polynomial approximation we have n functions (Taking in a vector of size n, returning a scalar) approximating.

The problem

Problem:
given the vector **b** with n data or function evaluations, and a nonlinear model function $\mathbf{R}(\mathbf{x})$, find the best **x** to approximate the given data.

$\mathbf{R}(\mathbf{x}) = \{r_1(\mathbf{x}), \ldots, r_n(\mathbf{x})\}^{\mathrm{T}}$, with $r_i : \mathbb{R}^m \to \mathbb{R}$.

$$\min_{\mathbf{x}\in\mathbb{R}^m} \Phi(\mathbf{x}) \quad \text{with} \quad \Phi(\mathbf{x}) = \frac{1}{2}\|\mathbf{R}(\mathbf{x}) - \mathbf{b}\|^2$$

Necessary minimum condition: $\nabla\Phi(\mathbf{x}^*) = \mathbf{0}$

- Find the critical point where the gradient is equal to zero

Phi has a special shape - which is good

- **Gradient:** $\nabla\Phi(\mathbf{x}) = J_{\mathbf{R}}(\mathbf{x})^{\mathrm{T}}(\mathbf{R}(\mathbf{x}) - \mathbf{b})$
- **Hessian matrix:** $H(\mathbf{x}) = J_{\mathbf{R}}(\mathbf{x})^{\mathrm{T}} J_{\mathbf{R}}(\mathbf{x}) + S(\mathbf{x})$

  with $S_{\ell j} = \sum_{i=1}^{n} \frac{\partial^2 r_i(\mathbf{x})}{\partial x_\ell x_j}(r_i(\mathbf{x}) - b_i), \qquad \ell, j = 1, \ldots, m$

- S(x) can be dropped as it is not super important

We use the Newton Method!

- Instead of using the exact hessian matrix we use the approximated hessian matrix

$$H \approx J_{\mathbf{R}}(\mathbf{x})^{\mathrm{T}} J_{\mathbf{R}}(\mathbf{x}),$$

Gauss-Newton algorithm:
INPUT: $\mathbf{x}^{(0)} \in \mathbb{R}^m$, $\mathbf{b} \in \mathbb{R}^n$, $J_{\mathbf{R}}(\mathbf{x})$, and $\mathbf{R}(\mathbf{x})$
FOR $k = 1, 2, \ldots$ until stop
  solve $J_{\mathbf{R}}(\mathbf{x}^{(k)})^{\mathrm{T}} J_{\mathbf{R}}(\mathbf{x}^{(k)})\mathbf{p} = J_{\mathbf{R}}(\mathbf{x}^{(k)})^{\mathrm{T}}(\mathbf{b} - \mathbf{R}(\mathbf{x}^{(k)}))$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$
END
OUTPUT: $\mathbf{x}^{(k+1)}$

```
tol = 10^-5;
itMax = 100;
x0 = [0;0]; % initial guess
fun = @(x) [ x(1)-.4; x(2)-.8 ; x(1)^2+x(2)^2-1 ];
jacobian = @(x) [ 1,0 ; 0,1 ; 2*x(1),2*x(2) ];  % define Jacobian fp(x)

[x, y, iter] = NonLinearLeastSquaresMethod(fun, jacobian, x0, tol, itMax)

function [x, y, iter] = NonLinearLeastSquaresMethod(Ffun, JacobianFun, x0, tol, itMax)
    iter = 0;
    err = tol + 1;
    x = x0; % first guess

    while (err >= tol & iter < itMax)
        J = JacobianFun(x);
        F = Ffun(x);
        delta = -J \ F;
        x = x + delta;

        err = norm(delta)
        iter = iter + 1;
    end
    y = norm(Ffun(x));

    %% Plot
    plot(x(1),x(2),'k*'); hold off %    mark solution point with '*'
    title('Solution of nonlinear least squares problem (*)')
end
```

```
tol = 10^-5;
itMax = 100;
x0 = [0;0]; % initial guess
fun = @(x) [ x(1)-.4; x(2)-.8 ; x(1)^2+x(2)^2-1 ];
jacobian = @(x) [ 1,0 ; 0,1 ; 2*x(1),2*x(2) ];  % define Jacobian fp(x)
b = [ 0.0375; 0.0750;-0.0429;]


[x, y, iter] = GaussNewtonMethod(fun, jacobian, x0,b, tol, itMax)

function [x, y, iter] = GaussNewtonMethod(Ffun, JacobianFun, x0,b, tol, itMax)
    iter = 0;
    err = tol + 1;
    x = x0; % first guess

    while (err >= tol & iter < itMax)
        J = JacobianFun(x)' * JacobianFun(x);
        F = JacobianFun(x)' * (b-Ffun(x));


        delta = J \ F;
        x = x + delta;

        err = norm(delta)
        iter = iter + 1;
    end
    y = norm(Ffun(x));

    %% Plot
    plot(x(1),x(2),'k*'); hold off %    mark solution point with '*'
    title('Solution of nonlinear least squares problem (*)')
end
```