AMATH 301 Final Cheat Sheet that I cannot even use on the final :)

Root Finding Methods (find roots of a function f(x) = 0)

MATLAB fzero(fun,x0) tries to find a point x where fun(x) = 0. This solution is where fun(x) *changes sign.* fzero cannot find a root of a function such as x^2.

Bisection Method: use midpoints
- Steps: make a left and right point, find the ***midpoint***, use for loop and find f(current_mid), the value of the function at the current x value.
- if (***abs(f_mid) < tol)*** is approximately equal to zero → root found, break;
- else if (***f_mid * sin(left) < 0***) → set right = mid (the root is in between the left and mid), and vice versa
  - **Bisection Cons**: only works if there is ***one root within the given starting range***, is somewhat slower than Newton's Method.
  - **Pro:** ***will always find root*** if exists in between starting points

Newton's Method: use derivatives
Steps: choose an initial point, set functions and find derivative of function
f = @ (x) x+0.5*sin(2*x)-3; fprime = @ (x) 1 + cos(2*x);
Declare x(k+1) = x(k) - f(x(k))/fprime(x(k));
if abs(f(x(k+1))) < tolerance, break from loop

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

Cons: need to choose a ***good starting point, need to know the derivative of the function, will NOT converge if on a critical point!!!***
Pros: ***faster than Bisection*** Method if converges

Linear Systems

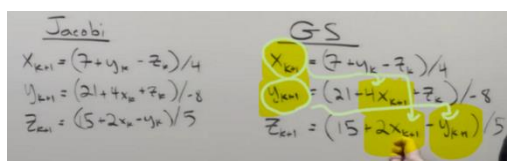
Code for Jacobi:
D = diag(diag(A));
T = A-D;
M = -D\T; c = D\b;
for k = 1:max
x_next = **M*x + c**;
if **error < tol** → break

eigenvalue **|λ| < 1**, the linear solving method will be ***guaranteed to converge***.
Choose to use **backslash** over inverse (inv(A)) in most cases.

condition number: This tells you ***how much solving a linear system will magnify any noise*** in your data. Will never fall below 1. An orthogonal matrix will not

Gaussian Elimination, a slow way of calculating solution for linear system, has an order of growth **O(N^3)** ("full price"). Ideal runtime for any iterative method: O(NlogN)
Jacobi Iteration: isolate all variables in the linear system
xk+1 = (-15+yk+5zk)/2
yk+1 = (4xk + zk + 21)/8
zk+1 = (7-4x+y)
Gauss-Seidel: enhancement of Jacobi Iteration. Use if you want a method that takes one input and does the whole process instead of individually filling out each variable like Jacobi.

A = S+T; % S is the lower triangle of A
% T is everything else



$$M = -S^{-1}T, \ c = S^{-1}b,$$

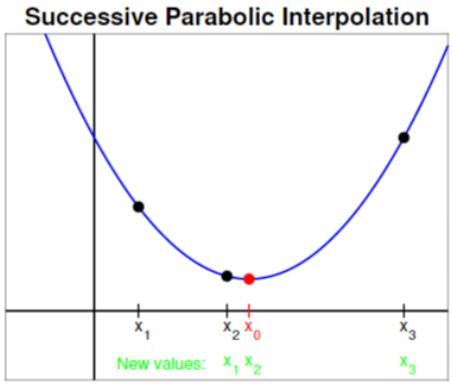
```
while norm(x_next - x_current, Inf) >= tol && iter < maxIters
x_current = x_next;
x_next = M*x_current + c;
```

- **Gauss-Seidel Cons:** need solutions from earlier lines, ***cannot be ran as parallel process (like computer with many processing cores)***
- **Pros:** much faster than Jacobi Iteration

Strictly Diagonally Dominant: all diagonal elements are strictly larger than the sum of the elements in the same row (of the matrix A). ***Jacobi and Gauss-Seidel methods will be guaranteed to converge if A is SDD***. SOR method may converge regardless.
Error checking condition for iterative linear solving methods:

- $\|Ax_k - b\|$
- $\|x_k - \overline{x}\|$
- $\|x_{k+1} - x_k\|$

LUP factorization: A = L*U; Therefore, L*U*x = b → y = L\b; x = y\b;

amplify any noise in your data. **Should I use an *iterative method over a direct* one?** → ***Large and sparse*** matrix A	- Why use **LUP** (Lower Triangle, Upper Triangle, Permutation) factorization over Gaussian Elimination? In many engineering applications, when you solve Ax=b, the matrix A remains unchanged, while the ***right hand side vector b keeps changing.*** - This decouples the factorization phase (usually computationally expensive) from the actual solving phase.
Min & Max Derivative-Free Methods (Optimization) **feasibility** (of regions) constrained = narrow feasibility, unconstrained = can take in any number **fminbnd:** a combination of Golden Section Search and SOR optimized to find the min value - Only for ***one-dimensional function*** over a specific domain	**Golden Section Search**: does not use any derivatives, function MUST BE ***UNIMODAL***!!! Is a lot like Bisection Method. **Has variables a, b, x1, x2.** **Successive Parabolic Interpolation:** pick three points, draw a curve through them.  $$x_0 = \frac{x_1 + x_2}{2} - \frac{(f(x_2) - f(x_1))(x_3 - x_1)(x_3 - x_2)}{2[(x_2 - x_1)(f(x_3) - f(x_2)) - (f(x_2) - f(x_1))(x_3 - x_2)]}$$ Get rid of side that is greater than x0. Ex: If x0 < x2 → x1new = x1old; If x2new = x0; x3new = x2old; - **Cons:** ***Sometimes does not converge***. Less reliable than Golden Section Search. - **Pros:** SOR can converge in fewer iterations, is ***faster than GSS***.
Gradient Descent: method that can find the minimum of functions of multiple variables. **fminsearch**(fun,x0): finds a single vector of values that will minimize a multi-dimensional function given some initial guess	**<u>Steps for Gradient Descent</u>** Outside the loop: f = @(p) fxy(p(1),p(2)); Define the objective function with one vector input. **fgrad = @(p) fgradxy(p(1),p(2));** Define gradient with one vector input. p = [-2;0]; Initial guess Inside the loop: **phi = @(t) p – t*fgrad(p);** Define line that points in the direction of steepest descent. **f_of_phi = @(t) f(phi(t));** Function handle defining value along path. grad = fgrad(p); Create a vector pointing in the direction, f increases the fastest. **tmin = fminbnd(f_of_phi, 0, 1);** Find the time it takes to reach min height on path. p = phi(tmin); Gives the point on the path that is the updated guess for the minimum. Exam Question: What is **domain and range of the minimum finding process of gradient descent**? § **R --> R^2, phi's codomain must be R^2, fgrad is R^2 to R^2** Which of the following are valid operations? **fgrad(phi(t))** is okay (fgrad needs an R^2 input, phi outputs an R^2) **phi(f(p))** f spits out something in R, phi wants R as domain

Curve Fitting

mu = center of a graph (change mu to move right or left)
- bigger mu = more to the right

sigma2 = height of peak, change to move up or down)
ex: sigma2 = 0.2 (very high), sigma2 = 1 (flatter curve)

Interpolation: getting data point where you don't have a data point (within range of current data)

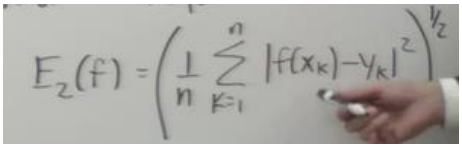
Polynomial Wiggle: middle part is accurate, but left side and right side are very inaccurate)

Find value of polynomial at x = 2: y = polyval(polynomial ,2);
An n-degree polynomial will go through n+1 points; Therefore, the error for a 2nd degree polynomial fit of 3 points will always be zero.

Maximal Error
- Weakness: ***can't apply to huge outliers*** (trying to minimize length of outlier to line), usually inaccurate in real life data
- Strengths: ***fast*** if points generally not outliers

Average Error
- Strength: can handle outliers very well, generally the safest method.
- Notes: **best accuracy if worried about worst case scenario, want to match that very well (try to ignore outliers)**

Root-Mean Square



- Strength: **can be *thrown off by outliers. but not as much as maximal error***

Least Square Fit
The minimum is from the inside summation. **No max error, only min error**!

Sum of Squared Errors
y = @(x) blah using mu and sigma
err = sum(abs(y(X)-Y).^2);
- ***this is generally well-fitted but ok with including outliers***

Exam Question: To fit a polynomial through n+1 data points, what degree must the polynomial be? **N degree (see below polynomial equation)**

$$y = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Numerical Difference Schemes (solving an ODE)
- error always increases over time

Error Order of Growth: **Forward and Backward** have an error growth of **O(time step)**
- Central and Leapfrog are second order accurate.
- **RK4 is fourth order accurate**

Second derivative error order of growth = **O(step^2)**

Taylor Series Expansion: The following equations are valid.

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \ldots$$

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots$$

Forward Euler
- **Explicitly solves for solution.** If you know the current point, just plug in Xk.
- UNSTABLE, if the step size is too big, the error will suddenly explode
- Otherwise usually **over-approximating** the actual solution

Backward Euler: (f(x) - f(x-dx))/dx; %% (rise over run but backwards)
- yk+1 = yk + delta t * **f(yk+1)**
- **implicit** method value being determined is a function of unknown variables, need to solve an algebraic equation for the unknown, **is stable**
- can also be written as: **xk+1 = inv(I-step*A) * xk;** when solving v = x' problem.

Central Difference: (f(x+dx) - (f(x-dx)))/(2*dx); %% middle derivative
- is the slope between the two lines representing forward and backward
- only has odd deltas, the even ones cancel out

Leap Frog
A = [0 1; -g/L -ro]; % matrix representing change to x and v every step
x0 = [theta0; v0];

Local Error vs Global Error: If solving for 0<t<T, you take order 1/step steps. Then local error = **O(step^n)**, then global error = **O(step^(n-1))**
 	
Phase Portraits: lets you see how state variables depend on each other.
- May show "limit cycle" (predator prey reliance)
	x1 = x0 + step*A*x0; % one step of forward to calculate x at first step
	for t = 2*step: step: 60
	  x_new = x0 + 2*step*A*x1; % x0 is two before, x1 is previous
	  x0 = x1;
	  x1 = x_new;
	end
	
	<u>Using ode45</u>
	ts = 0:step:T; %% a range of times
	Z0 = [theta0 ; v0];
	odefun2 = @(t,Z) [dxdt2(Z(1),Z(2));
	dvdt2(Z(1),Z(2))]; %% t is ts, Z is initial values
	[t,isol] = ode45(odefun2,ts,Z0);
	- Note: ode45 becomes unstable under "stiff differential equations" situation
	
	<u>Runge-Kutta ODE</u>:
	RK2 (midpoint): 2nd order of growth error → more accurate than forward and backward.
	for k = 1: N-1; %% from first point to almost last point
	  k1 = f(x(k)); %% x value at k
	  k2 = f(x(k) + dt/2 * k1);
	  x(k+1) = x(k) + dt*k2;
	end
	
	RK4: even more accurate: Do half step forward, do another half step of forward from first point, use a weighted average of the 4 slopes and take a full step
	x(k+1) = x(k) + dt/6 * (k1 + 2*k2 + 2*k3 + *k4); %% average of all k's

Numerical Integration MATLAB's built-in integral function: works only for function handles **integral**(f, 0, 8);	**<u>Left and Right rectangle rule</u> <u>(Global error: O(step))</u>** left_rect = dt*sum(c(1:end-1)) right_rect = dt*sum(c(2:end)) **<u>Midpoint Rule</u>** - Global error: **O(step^2)** **<u>Trapezoidal rule:</u> <u>Global error = O(step^2)**</u> trap = **dt/2*(c(1) + c(end) + 2*sum(c(2:end-1));** **<u>Simpson's Rule</u>** → closest to true solution, **Global error = O(step^4)** simpson = **dt/3** * (c(1) + c(end) + **4*sum(c(2:2:end-1))** + **2*sum(c(3:2:end-2))**; - The first and last endpoints, and 4*the even points, 2*odd points - must have **EVEN NUMBER OF INTERVALS!!!**