, always watching....

We are one

Line Search Methods

Exact

Finds local minimum along a direction

Bisection (bracketing) - Finds the root in a given range (bracket) by sampling the midpoint. Robust but slow

Newton’s method - uses gradient info, faster but requires gradient info

Brent’s method - combines bisection with newton’s. Generally best for 1D root finding - Golden Section + Polynomial

Inexact

Finds point along a direction that satisfies Wolfe’s Conditions (a step to a point that is “good enough”)

Wolfe Conditions:

$$f(x_k + \alpha p_k) \leq f(x_k) + \mu_1 \alpha g_k^T p_k$$

Sufficient Decrease - objective has sufficiently decreased

K iterations

P direction

Alpha magnitude

Mu usually around 10^-4

New point must be less than last

Curvature Condition (not too small of a step) - slope has sufficiently decreased

Mu usually 0.9

Slope at new point must be less negative

$$g(x_k + \alpha p_k)^T p_k \geq \mu_2 g_k^T p_k$$

Strong Wolfe Conditions (0 < μ1 < μ2 < 1)

Both are satisfied

Backtracking - simple

Check sufficient decrease, decrease alpha by some value rho (e.g. 0.5)

Gradient Based Optimization

Gradient vs Hessian

Gradient is first derivative (slope), you’ll have as many gradients as design variables

Hessian is second deriv. (curvature), n x n where n = # design variables

Gradient is defined as positive, that is why steepest descent is -g/norm(g)

Optimality Conditions

abs(gradient) = 0

Hessian is positive definite - positive curvature

Unconstrained Optimization

Initial guess

Compute search direction (P)

Find alpha that satisfies Wolfe Conditions

Update the design variables: x_k+1 = x_k + $\alpha_k p_k$

Methods to get search direction (P):

Steepest Descent

Easy

Very poor, low convergence, orthogonal zig zag, doesn't naturally pick a step size, just direction

Conjugate Gradient

Uses gradient history (steepest route with memory)

Better, but still not great

Newton

Uses second order information to do what?

To find step direction H_k * p_k = -g_k, can also give you step size, as it fits quadratic function to point, but you don't need to go there

Requires a hessian, which can be difficult/impossible

Quasi-Newton

Use gradients to estimate hessian

Secant rule: update hessian on every iteration, rather than re-compute

Hessian might not be positive definite -- we really want the inverse (V) instead (BFGS?), advantage over Newton method, do not need to invert hessian

Trust regions

Find step size and then perform approximate minimization in region- don't need to worry about backtracking within the trust region.

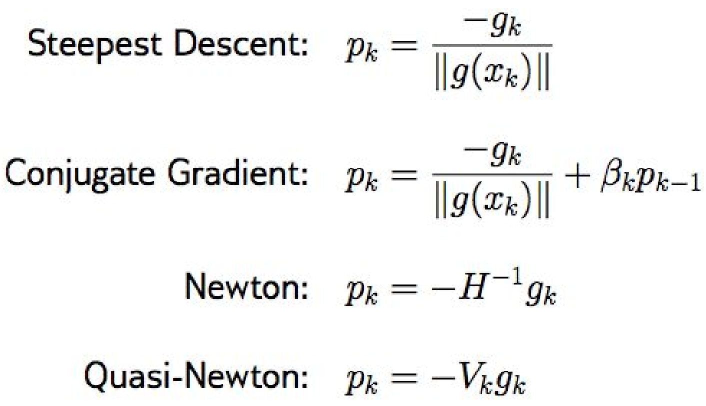
More on this?
Pick a maximum step size first - this is your trust region, then a direction and step - usually done using local quadratic model
Accuracy of method is determined by actual decrease / predicted decrease from your step
If accurate, ratio near 1, increase trust (maximum step size allowed)
If inaccurate, decrease trust (limit maximum step size)
Trust region near solution should be large, meaning the local quadratic model is accurate, so it won’t be taking large steps anyway

DFP

Estimates H and then inverts it to find step - How is this different from quasi newton? Or is this a QN method?
The primary advantage is that we don’t have to compute the exact Hessian at each point, which may be computationally expensive

BFGS

Estimates H^-1 to find step, do not need to invert hessian
Always positive definite



Steepest Descent: $p_k = \dfrac{-g_k}{\|g(x_k)\|}$

Conjugate Gradient: $p_k = \dfrac{-g_k}{\|g(x_k)\|} + \beta_k p_{k-1}$

Newton: $p_k = -H^{-1} g_k$

Quasi-Newton: $p_k = -V_k g_k$

Obtaining Gradients

Need n x 1 gradients, m x n gradients of constraints (m constraints)

Symbolic Gradients

Just like in MATH 112….
Advantage: exact, quick to compute
Disadvantage: only works for simple functions

Finite Difference

Step size: 1e-6 (for O(1)); scale inputs to O(1); can’t really go smaller because of subtraction approximation error, too large leads to truncation error

For a well-scaled problem we can show that the best h is approximately the square root of the relative error of f
Forward, backward, central approximation

$$\boxed{f'(x) \approx \frac{f(x+h) - f(x)}{h}}$$

Advantage: easy, black-box (don't have to enter function)
Disadvantage: inexact, lots of function calls

Complex Step

$$\boxed{\frac{\partial f}{\partial x_k} \approx \frac{\operatorname{Im}\left[f\left(x + ih\hat{e}_k\right)\right]}{h}}$$

Step size(h): 1e-20; no subtraction approximation (subtractive cancellation) error
Similar to finite difference, but with complex step

Advantage: exact
Disadvantage: must get into code, make complex numbers possible in code; requires n function evaluations but each evaluation is twice as expensive. Computing gradients is slower! But convergence is likely faster. Not effective for large problems or anything with matrix maths.

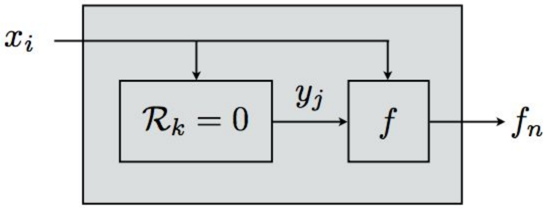
Automatic Differentiation

Breaking function up into simple operators
Overloading
Source code transformation - faster but requires updating with every code update

Advantage: exact; scales well; much less function calls; much faster than finite difference / complex step
Disadvantage: must get into code, takes more time to set up

Adjoint

Uses only partials



Adjoint: lots of inputs (phi first)
- More efficient than Direct method when n_x>n_f

Direct: lots of outputs (psi first)
- More efficient than Adjoint method when n_f>n_x

$$\frac{df}{dx} = \frac{\delta f}{\delta x} - \frac{\delta f}{\delta y}\left[\frac{\delta R}{\delta y}\right]^{-1}\frac{\delta R}{\delta x}$$

$$\frac{df_n}{dx_i} = \frac{\partial f_n}{\partial x_i} - \underbrace{\frac{\partial f_n}{\partial y_j}\overbrace{\left[\frac{\partial \mathcal{R}_k}{\partial y_j}\right]^{-1}\frac{\partial \mathcal{R}_k}{\partial x_i}}^{\Phi}}_{\Psi}$$

Advantage: exact, good for complex functions, independent of number of design variables
Disadvantage: have to get partials

Constrained Optimization

Constraints

C*lambda = 0 always

Active: solution touching constraint c=0, lambda>0
Inactive: solution not touching constraint c<0, lambda=0

Function gradient and constraint gradient must be parallel $\nabla f(x^*) = -\lambda \nabla c(x^*)$

Lagrangian

$$\mathcal{L}(x, \lambda) = f(x) + \lambda \hat{c}(x)$$

Turn constrained problem into an unconstrained problem
Bundles gradients of function and constraints into a single equation that can then be optimized
Now gradient of L must be 0 (optimality criteria)
(set gradient=0, take partials, sum them. same with constraints, put it all together. function value * multiplier * constraint value.)

KKT Conditions

$$\nabla_x \mathcal{L}(x, \lambda) = 0$$
$$\hat{c}(x) = 0$$
$$c(x) \leq 0$$
$$\lambda \geq 0$$
$$\lambda c(x) = 0$$

Optimality conditions for constrained optimization
Karush Kahn Tucker
Partials of lagrangian are zero
Second-order sufficient conditions

Slack variables

Used for inequality variables (to turn into equality)

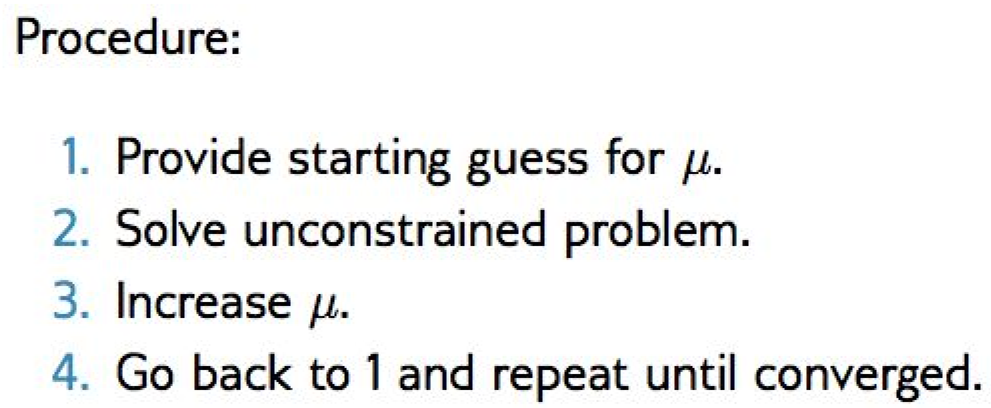
c(x) <= 0; c(x) + s^2 = 0; c(x) = -s^2

Complementarity conditions: lambda*s = 0 (two cases: s = 0 (active), s > 0 (inactive))
Now we just have to take a bunch of partial derivatives, solve linearly

Penalty Methods

Minimize an unconstrained problem where the infeasibility is minimized along with the objective
Algorithm: check termination conditions, minimize penalty function, increase penalty parameter
Convert constrained problem into unconstrained
Simple
Always infeasible, ill conditioning which means? - Function space has large spikes and solution is almost always partially in the infeasible range. Mu needs to go to infinity to have perfectly feasible solution.

$$F(x) = f(x) + \mu P(x)$$



Procedure:

1. Provide starting guess for μ.
2. Solve unconstrained problem.
3. Increase μ.
4. Go back to 1 and repeat until converged.

Exterior Methods

Don’t start affecting the problem until boundary is crossed
Quadratic Penalty Method
Augmented Lagrangian
Quadratic Program (QP): an optimization problem with a quadratic objective function and linear constraints. Increases feasible space over time

Sequential Quadratic Programming (state of the art, faster on smaller problems, compared to interior point methods)

> iteratively minimizing a local quadratic approximation of the Lagrangian with a local linear approximation of the constraints **shape your space w/ constraints**
> Can be done with inequality constraints, using slack variables
> Need to keep track of active set of constraints :(
> **Merit Functions? What exactly are they?**

Interior Methods (state of the art, faster on larger problems)

Advantage: never break constraint
Disadvantage: must start inside feasible space

Logarithmic Barrier Method: implemented iteratively by increasing mu value
$\pi(x) = f(x) - \mu log(c(x))$ ←- old method, new is log(slack variable)
Inverse Barrier Method: sum of inverse of constraints
Mu goes to 0

Multi-Objective functions

Is there a higher-level objective that you are actually after? Are some of your objectives actually constraints?
Explore tradeoffs between potential objectives
Domination (whichever point is less than the other in terms of all variables) → pareto front
How?
- Combine objectives as weighted sum
  - Simple, but difficult to determine weighting
- Constraint epsilon: set other objectives as constraints, vary their limits
  - Simple, more intuitive, but there’s non-uniform spacing

Gradient Free Optimization

Does not care what the function looks like or where the constraints are; simply samples the entire design space, communicates between points and figures out the space itself
Better for non-continuous, data/noisy, can’t get reliable derivatives, many local minima

Nelder Mead

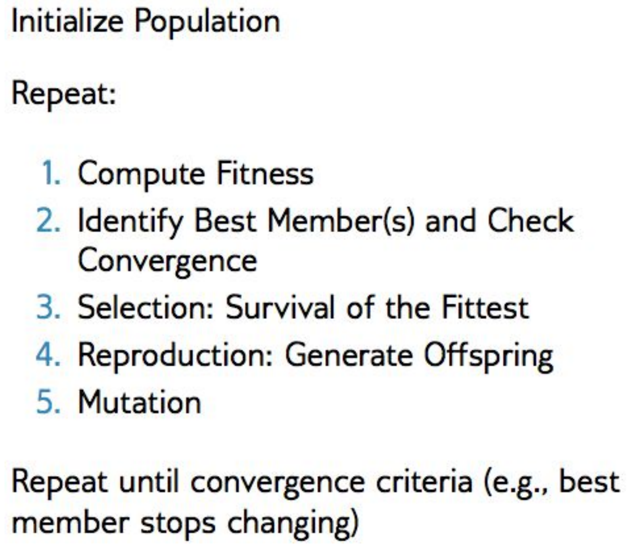
Create simplex
Identify: best, lousy, worst points
Simple, inaccurate/expensive

Genetic Algorithm

An evolutionary algorithm
Initialize, Reproduction, Mutation

Methods for determining mating pool: tournament, roulette
Reproduction: crossover, interpolation, extrapolation, pass on bits, etc.
Mutation: add randomness to offspring (explore or exploit)

Converge when best doesn’t change for 10 iterations



Initialize Population

Repeat:

1. Compute Fitness
2. Identify Best Member(s) and Check Convergence
3. Selection: Survival of the Fittest
4. Reproduction: Generate Offspring
5. Mutation

Repeat until convergence criteria (e.g., best member stops changing)

Initial pop. usually at least 10x number of design variables

• Probabilistic. Will get different results each time. May need to run multiple times.
• Population-based rather than point-based. **Suited for multiobjective optimization**, and exploration.
• Real or binary encoding. **Can handle discrete variables(binary encoded only).**
• Easily parallelizable. • Heuristic, with many variations.

Particle Swarm

Each has momentum, memory, and social influence

1. Initialize positions (x) and velocities (v)x
2. Evaluate function values f(x)
3. Update best position for particles (p_i) and for the whole swarm (p_g)
4. Update velocities and positions: enforce bounds
5. Repeat Steps 2-4 until convergence
    a. Best fitness hasn’t changed for many iterations
    b. Mean of the particle positions (p_i) hasn’t changed for several iterations
    c. Velocity of particles falls below tolerance for several iterations
    d. Distance between particles and best position (p_g) is below tolerance
    e. Distance between best and worst fitness is below tolerance

Parameters to choose: self-confidence, swarm-confidence, inertial parameter, randomness

Velocity clamping: we put an upper bound on the velocity so that we don’t move too far. Δx of around 10–20% of range is reasonable.

Velocity damping: at each iteration we decrease what that maximum bound is (e.g., Vmax = 0.9Vmax), or decrease the inertial parameter w, to improve convergence.

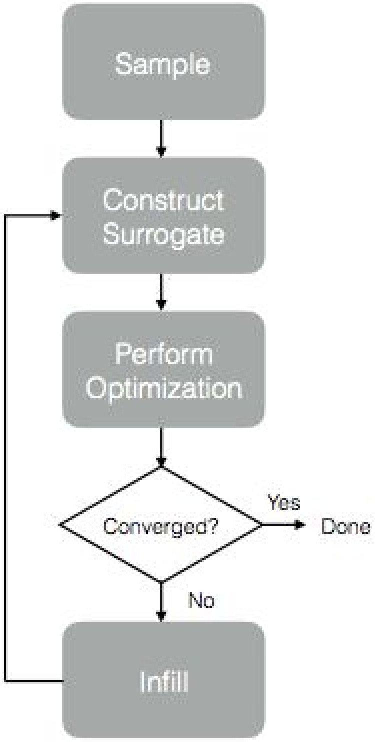
Integer Programming

We used excel..?
Salesman route problems. Very hard to solve. Cannot be solved with brute force. We can usually only solve Linear Integer Programming problems.

Surrogate-Based Optimization (SBO)

For when simulation is expensive, output is noisy, experimental data, understand functional relationships

- get a sample from function
- create surrogate model w/ optimum
- run function at that point
- re-create surrogate with that point going to the correct value (infill)
- re-evaluate, get optimum, etc.



Latin hypercube sampling:

is a statistical method for generating a sample of plausible collections of parameter values from a multidimensional distribution.

Optimization Under Uncertainty

Variability in engineering
• Robust: performance is less sensitive to inherent variability (objective function)
• Reliable: less prone to failure under inherent variability (constraints)

Once you've optimized: Monte Carlo Simulation (propagates uncertainties, check reliability)

1. Sample N points from input probability distribution (xi). (add some variation)
2. Evaluate output fi = f(xi).
3. Compute statistics on fi .

100,000 function calls
Black box, efficient, doesn't scale well

Worst Case:

1. Compute deterministic optimum (what you've already been doing).

2. Estimate worst-case Δc at the deterministic optimum.
3. Adjust constraint to $c(x) + \Delta c \le 0$ and reoptimize (start from the solution you just found).

Transmitted variance functions the same, but assumes probability distributions known