Lecture 2: Unconstrained Optimization

Kevin Carlberg

Stanford University

July 28, 2009

Unconstrained optimization

- This lecture considers unconstrained optimization

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad f(x)$$

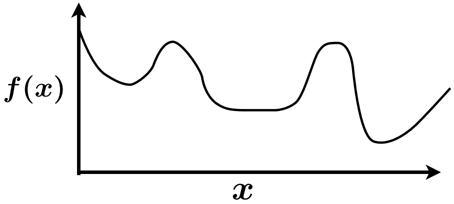
- Things become significantly more complicated with constraints!

Univariate minimization

- Consider the unconstrained minimization of a function in one dimension

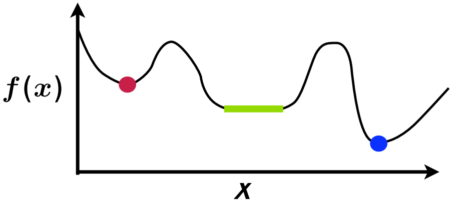
$$\underset{x\in\mathbb{R}}{\text{minimize}} \quad f(x) \tag{1}$$

- In this class, we assume all functions are "sufficiently smooth" (twice-continuously differentiable)



- What is a solution to (1)?

What is a solution?

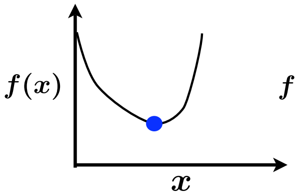


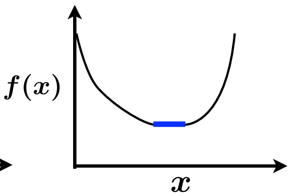
- Global minimum: A point x^* satisfying $f(x^*) \leq f(x)\ \forall x \in \mathbb{R}$
- Strong local minimum: A neighborhood $\mathcal{N}$ of x^* exists such that $f(x^*) < f(x)\ \forall x \in \mathcal{N}$.
- Weak local minima A a neighborhood $\mathcal{N}$ of x^* exists such that $f(x^*) \leq f(x)\ \forall x \in \mathcal{N}$.

Convexity

- For convex objective functions in one variable,

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$





- In this case, any local minimum is a global minimum!

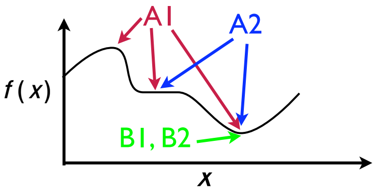
Optimality conditions for univariate minimization

Theorem (Necessary conditions for a weak local minimum)

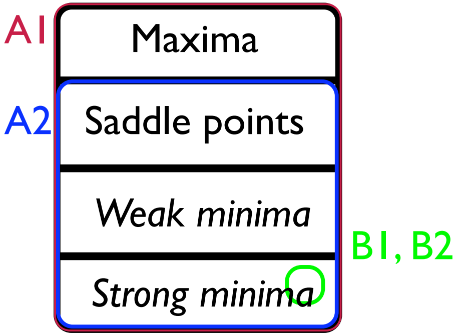
A1. $f'(x^*) = 0$ *(stationary point)*
A2. $f''(x^*) \geq 0$.

Theorem (Sufficient conditions for a strong local minimum)

B1. $f'(x^*) = 0$ *(stationary point) and*
B2. $f''(x^*) > 0$.



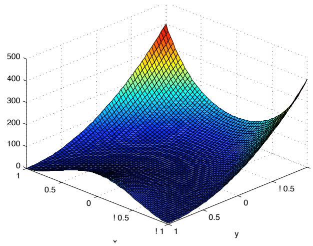
Optimality conditions for univariate minimization



Multivariate minimization

- Now, consider the unconstrained minimization of a twice-continuously differentiable function in n dimensions

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) \tag{2}$$



- The same notions of weak local, strong local, and global minima, as well as convexity, extend to multiple dimensions.

Stationary points

Definition (Stationary point)

A stationary point x^* of the function f is any point satisfying $\nabla f(x^*) = 0$.

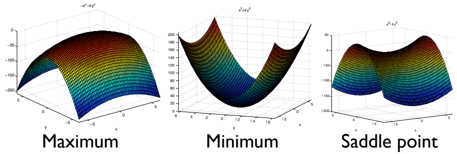


Figure: Types of stationary points in multi-dimensions

Optimality conditions for multivariate minimization

In multiple dimensions, the conditions are simply the multivariate extensions of the univariate conditions

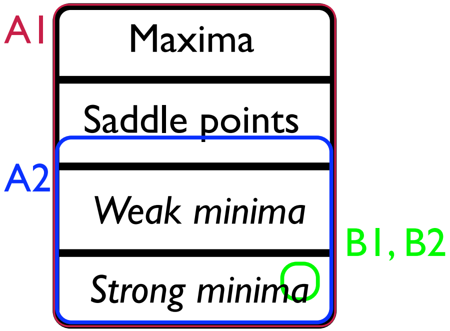
Theorem (Necessary conditions for a weak local minimum)

A1. $\nabla f(x^*) = 0$ *(stationary point)*
A2. $\nabla^2 f(x^*)$ *is positive semi-definite* $(p^T \nabla^2 f(x^*) p \geq 0$ *for all* $p \neq 0)$

Theorem (Sufficient conditions for a strong local minimum)

B1. $\nabla f(x^*) = 0$ *(stationary point)*
B2. $\nabla^2 f(x^*) > 0$ *is positive definite* $(p^T \nabla^2 f(x^*) p > 0$ *for all* $p \neq 0)$.

Optimality conditions for multivariate minimization



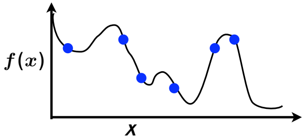
Optimization algorithms

- We now know what a mathematical optimization problem is, and we can characterize local and global solutions using the optimality conditions.
- How do we *compute* these solutions?
  - **Analytically**: only possible for some simple problems (e.g. Brachistochrone problem, univariate minimization)
  - **Numerically**: required for most engineering optimization problems (too large and complex to solve analytically)

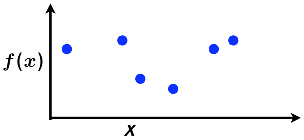
  $\longrightarrow$ Numerical optimization algorithms are used to numerically solve these problems with computers

Optimization algorithms

- In general, we are *mostly blind* to the function we are trying to minimize. We can only compute the function f at a finite number of points, and each evaluation is expensive



True function



Observed function

- Higher-order information (gradient ∇f and Hessian $\nabla^2 f$) is sometimes available at these points, but is more expensive to compute

Optimization algorithms

- Goals
  - **Robust**: low failure rate, convergence conditions are met
  - **Fast**: convergence in a few iterations and low cost per iteration
  - **Feasible**: reasonable memory requirements
- Algorithm design involves tradeoffs to achieve these goals (e.g. using high-order information may lead to fewer iterations, but each iteration becomes more expensive)
- Algorithms are iterative in nature
- Categorization
  - **Gradient-based** v. **Derivative-free**
  - **Global** v. **local**
  - Gradient-based algorithms tend to be local, while derivative-free algorithms tend to be global

Gradient-based algorithms

- Imagine you are lost on a mountain in extremely thick fog

- How would you get down?
- Chances are, you would use the *slope* of the ground beneath you in some way to go downhill and descend the mountain
- This is the approach taken by gradient-based algorithms

Benefits and drawbacks of gradient-based algorithms

- Benefits
  - Efficient for many variables
  - Well-suited for smooth objective and constraint functions
  - Efficient computation of gradients possible for PDE-constrained problems
- Drawbacks
  - Convergence is only local

  $\longrightarrow$ Mitigated by using multiple starting points to find multiple local minima, and hopefully the global minimum
  - Not well-suited for discrete optimization

  $\longrightarrow$ Mitigated by reformulating discrete problems as continuous (e.g. branch and bound methods)

Framework

- Gradient-based methods compute both a **direction** p_k and a **step length** α_k at each iteration k

Algorithm 1 Gradient-based framework

```
Choose initial variables $x_0$, $k = 0$
while (not converged) do
    $\boxed{\text{Choose direction } p_k \text{ and step length } \alpha_k}$
    $x_{k+1} = x_k + \alpha_k p_k$
    $k \leftarrow k + 1$
end while
```

- **Line search methods**: 1) compute p_k, 2) compute α_k
- **Trust region methods**: 1) compute a maximum step length, 2) compute p_k and actual step length α_k

Theorem (Sufficient conditions for global convergence)

For sufficiently smooth, well-defined problems, sufficient conditions for global convergence $\lim_{k\to\infty} \|\nabla f_k\| = 0$ of line search methods are:

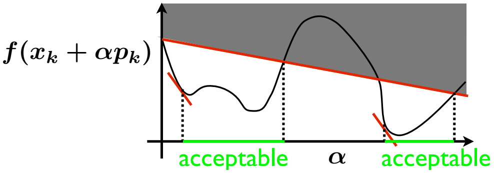
C1. p_k are descent directions ($p_k^T \nabla f(x_k) < 0$)

C2. α_k produce a sufficient decrease (satisfy the Wolfe conditions)

- C2. Wolfe conditions ($0 < c_1 < c_2 < 1$):

$$\text{Decrease } f: \quad f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k,$$

$$\text{Increase } \nabla f: \quad \nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k.$$



1. Choose p_k that is a descent direction (C1)
   - Steepest descent: First-order, linear convergence
   - Conjugate gradient: First-order, linear (faster) convergence
   - Newton: Second-order, quadratic convergence
   - Quasi-Newton: First-order to approximate second-order, superlinear convergence
2. Choose step length α_k satisfying the Wolfe conditions (C2)
   1. Bracketing: find an interval containing a good step length
   2. Bisection/interpolation: compute a good step in this interval

1. **Choose p_k that is a descent direction (C1)**
   - Steepest descent: First-order, linear convergence
   - Conjugate gradient: First-order, linear (faster) convergence
   - Newton: Second-order, quadratic convergence
   - Quasi-Newton: First-order to approximate second-order, superlinear convergence
2. Choose step length α_k satisfying the Wolfe conditions (C2)
   1. Bracketing: find an interval containing a good step length
   2. Bisection/interpolation: compute a good step in this interval

Steepest descent

- This is what you would likely do if stranded on the mountain
- Steepest descent chooses the fastest downhill direction

$$p_k = -\nabla f(x_k)$$

- **Advantages**: only first-order information is required, always a descent direction, low storage
- **Disadvantages**: slow on difficult problems, sensitive to scaling

Conjugate gradient (CG)

- CG methods "correct" the steepest descent direction

$$p_k = -\nabla f(x_k) + \beta_k p_{k-1}$$

- β_k computed to make p_k and p_{k-1} (approximately) conjugate, which allows the method to (better) account for previous progress
- **Advantages**: more effective than steepest descent and almost as simple to implement, only first-order information is required, low storage
- **Disadvantages**: moderate convergence rate, sensitive to scaling

Newton

Theorem (Taylor's theorem)

For f twice-continuously differentiable,

$$\nabla f(x+p) = \nabla f(x) + \nabla^2 f(x)p + \int_0^1 \left[\nabla^2 f(x+tp) - \nabla^2 f(x)\right] p dt$$

- By setting $\nabla f(x+p) = 0$ (want a stationary point) and ignoring the $o(\|p\|)$ integral, we can solve for p_k

$$p_k = -(\nabla^2 f(x))^{-1} \nabla f(x)$$

- **Advantages**: Quadratic convergence, natural step length, insensitive to scaling
- **Disadvantages**: Requires second-order information, may not be a descent direction, must store Hessians

Newton's method with Hessian modification

- If $\nabla^2 f(x_k)$ is not positive definite, the Newton step may not be a descent direction

$$p_k^T \nabla f(x_k) = -\nabla f(x_k)^T \left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k)$$

- Since descent directions (and Wolfe condition satisfaction) are sufficient for global convergence, we would like to use a modified Hessian that is positive definite.

Algorithm 2 Hessian modification

```
Choose initial variables $x_0$, $k = 0$
while (not converged) do
    Compute $B_k = \nabla^2 f(x_k) + E_k$, where $E_k$ makes $B_k$ sufficiently positive definite
    Solve $B_k p_k = -\nabla f(x_k)$
    Compute $\alpha_k$ to satisfy the Wolfe conditions
    $x_{k+1} = x_k + \alpha_k p_k$, $k \leftarrow k + 1$
end while
```

Quasi-Newton

- In many cases, it is not possible or too expensive to compute the exact Hessian $\nabla^2 f(x_k)$
- Quasi-Newton methods approximate the Hessian by some matrix $B_k \approx \nabla^2 f(x_k)$
- B_k is updated at each iteration using only first-order information
- **Advantages**: Superlinear convergence, no second-order information explicitly calculated, natural step length, insensitive to scaling
- **Disadvantages**: may not be a descent direction, approximate Hessians may not be accurate, approximate Hessians can be dense even if true Hessian is sparse (bad for large-scale problems)

Quasi-Newton updates

- **Secant condition**: should be satisfied by the update to B_k (from Taylor's theorem neglecting the $o(\|p\|)$ integral term)

$$B_{k+1}\left(x_{k+1} - x_k\right) = \nabla f_{k+1} - \nabla f_k$$

- We can impose other conditions on B_k such as symmetry and positive definiteness

1. **Symmetric rank-one update (SR1)**: Enforce symmetry

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k}$$

2. **Broyden, Fletcher, Goldfarb, and Shanno (BFGS)**: Enforce symmetry, positive definiteness, rank-two update

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

1. Choose p_k that is a descent direction (C1)
    - Steepest descent: First-order, linear convergence
    - Conjugate gradient: First-order, linear (faster) convergence
    - Newton: Second-order, quadratic convergence
    - Quasi-Newton: First-order to approximate second-order, superlinear convergence
2. **Choose step length α_k satisfying the Wolfe conditions (C2)**
    1. Bracketing: find an interval containing a good step length
    2. Bisection/interpolation: compute a good step in this interval

Step-length selection

- Want to compute an α_k that satisfies the Wolfe conditions
- This amounts to an *inexact* line search
- Assume p_k is a descent direction

Step-length selection

1. Bracketing: find an interval containing a good step length
2. Bisection/interpolation: compute a good step in this interval
    1. Use only function values
        - Fibonacci search
        - Golden Section serach

        $\rightarrow$ less efficient, cannot verify Wolfe conditions
    2. Use function values and the gradient
        - Quadratic or cubic interpolation
        - A root-finding algorithm (find root of $\nabla f(x + \alpha_k \boldsymbol{p}_k)$)

        $\rightarrow$ more efficient, can verify Wolfe conditions

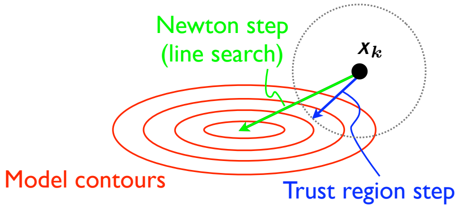
- Newton and Quasi-Newton methods have a “natural” step length $\alpha_k = 1$ (this minimizes convex, quadratic functions)
- p. 60–61 of Nocedal & Wright contains a good algorithm

Trust region methods

- Line search methods: 1) compute p_k, 2) compute α_k
- **Trust region methods**: 1) compute a maximum step length, 2) compute p_k and actual step length α_k

Trust region methods

- Trust regions define a region within which they *trust* the accuracy of a quadratic model, then minimize the model in this region



- If the step is unacceptable (inaccurate model), the size of the region is reduced (we trust the model less)

Trust region methods

- Trust region methods use a quadratic model $m_k(p)$ of the true function $f(x_k+p)$ at the point x_k
$$m_k(p) = f_k + g_k^T p + \frac{1}{2} p^T B_k p$$
- If B_k is the exact Hessian, the difference between $m_k(p)$ and $f(x_k+p)$ is $O(\|p\|^3)$
- At each trust-region step, the following constrained problem is approximately solved for p_k

$$\underset{p\in\mathbb{R}^n}{\text{minimize}}\ m_k(p) \quad \text{s.t. } \|p\| \le \Delta_k \tag{3}$$

Actual reduction to predicted reduction ratio

- The quality of $m_k(p)$ is assessed by the actual to predicted reduction ratio

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p_k)}$$

- ρ_k small: the model over-predicts the decrease in objective function (get worse-than-predicted answer)
- ρ_k large: the model under-predicts the decrease in objective function (get better-than-predicted answer)
- The logic employed by trust region methods says that an under-prediction is good, and we should trust our model more

Algorithm 3 Trust region

```
Choose initial variables $x_0$, $k = 0$
while (not converged) do
    Obtain $p_k$ by approximately solving Eq. (3)
    Evaluate the ratio $\rho_k$
    if $\rho_k < 1/4$ then
        Reject step $x_{k+1} = x_k$
        Shrink trust region (trust the model less)
    else if $1/4 \leq \rho_k \leq 3/4$ then
        Accept step $x_{k+1} = x_k + \alpha_k p_k$
    else
        Accept step $x_{k+1} = x_k + \alpha_k p_k$
        Grow the trust region (trust the model more)
    end if
    $k \leftarrow k + 1$
end while
```

Solving the quadratic subproblem

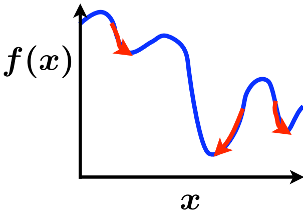
Algorithm 4 Trust region

```
Choose initial variables $x_0$, $k = 0$
while (not converged) do
    $\boxed{\text{Obtain } p_k \text{ by approximately solving Eq. (3)}}$
        ⋮
end while
```

- As in line search methods, the subproblem must only be solved approximately

Gradient-based algorithms for global optimization

- Gradient-based algorithms are best-suited for finding local optima because they "go downhill" until local optimality conditions are satisfied
- To find multiple local optima (and hopefully the global optimum), gradient-based methods can be run multiple times using different starting points that should be in different "basins of attraction"



Computation of gradients

- To implement gradient-based algorithms, derivative information must be computed
- There are three main ways to compute these gradients
  1. Analytical
  2. Finite differences
  3. Automatic differentiation

Finite differences

- We can approximate the gradient by evaluating the function several times when the gradient is unavailable analytically
- **Forward-difference**: 1st-order accurate

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x+\epsilon e_i) - f(x)}{\epsilon} + O(\epsilon)$$

- **Central-difference**: 2nd-order accurate, but twice as expensive

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x+\epsilon e_i) - f(x-\epsilon e_i)}{2\epsilon} + O(\epsilon^2)$$

- Challenge: ϵ too large $\rightarrow$ inaccurate, ϵ too small $\rightarrow$ subtractive cancellation due to round-off error

Automatic differentiation

- Use computational representation of a function
- Key observations:
  1. Any function is composed of a sequence of simple operations
  2. The chain rule from calculus. For $f(y(x(w)))$,

$$\frac{df}{dw} = \frac{df}{dy}\frac{dy}{dx}\frac{dx}{dw}$$

- Performs differentiation on only basic operations
- Avoids subtractive cancellation
- Software tools (e.g. ADIFOR) do this automatically

Why derivative-free algorithms?

- Gradients may not be available ($f(x)$ from experiment, impractical to code analytic gradients)
- Noise or non-smoothness in the objective function makes finite differences inaccurate
- May want to direct effort *globally* (more function evaluations at more points) rather than locally (more information at the same points)

Benefits and drawbacks of derivative-free algorithms

- Benefits
  - Well-suited for discrete variables
  - Often better at finding the global optimum
  - Robust with respect to function noise
  - Useful for multi-objective optimization
  - Amenable to parallel computing
- Drawbacks
  - Too expensive for many variables
  - Efficient treatment of general constraints difficult

Derivative-free algorithm categorization

1. Heuristic: use techniques inspired by nature (global)
   - Simulated annealing
   - Genetic algorithms
   - Swarm intelligence (particle swarm optimization, ant colony optimization)
2. Direct search: query a sequence of nearby points (local)
   - Directional: coordinate search, pattern search, generalized pattern search
   - Simplicial: Nelder-Mead nonlinear simplex
3. Line search: finite differences adapted to handle noise (local)
   - Implicit filtering
4. Surrogate-based optimization: use response surfaces (RS)

   $\longrightarrow$ RS types: Kriging, radial basis functions, neural networks
   1. Local: trust region model management, surrogate management framework
   2. Global: maximize expected or probability improvement, etc.

- Genetic Algorithms (GAs) were invented in the 1960's by John Holland, who wanted to better understand the evolution of life by computer simulation
- The algorithm is based on **reproduction** (crossover and mutation) and **selection** (survival of the fittest)

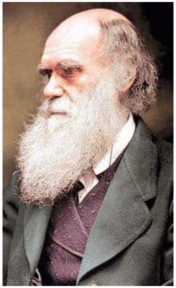


Figure: Charles Darwin

$$\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad f(x)$$

- A population member is represented by a point x in the variable space (it's DNA)
- "Fitness" is the objective function value $f(x)$
- Rather than work with a single point at a time, we consider an entire *population* of members at any given time
- Because the entire variable space is being constantly searched, the algorithm is more likely to find a global optimum and won't be "trapped" by local optima
- Unfortunately, very expensive for problems with many variables

Overview of genetic algorithm

1. Initialize population
2. Determine mating pool
3. Generate children via crossover
   - Continuous variables: interpolate
   - Discrete variables: replace parts of their representing variables
4. Mutation (add randomness to the children's variables)
5. Evaluate fitness of children
6. Replace worst parents with the children

Rest of the week

- We now know something about formulating, categorizing, and solving unconstrained minimization problems
- We next consider the inclusion of constraints:
    1. Constrained optimization
    2. PDE-constrained optimization