Lecture 10, Oct 11, 2023

Numerical Methods for Unconstrained Optimization

- Finding a global minimum is hard to do analytically because we need to find all stationary points, and then compute the value of the function at all stationary points and then the boundaries, which could be expensive
  - Often we will have to use a numerical root finding method to solve for where the gradient is zero
  - In this case it might be easier to solve for the minimum numerically to begin with

One-Dimensional Methods

- We will focus on finding the stationary points $\vec{\nabla} f(x^*) = 0$
- Newton's method can be used if the function is differentiable:
  - Approximate $f(x)$ locally by a quadratic: $f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2$
  - $f'(x) \approx f'(x_k) + f''(x_k)(x - x_k) \implies x_{k+1} = x_k - \dfrac{f'(x_k)}{f''(x_k)}$
    * Keep iterating until the gradient is below some tolerance
    * Once we find a stationary point, check if it is a minimum using the Hessian/second derivative
  - This is equivalent to applying Newton's method for root finding to solve for $f'(x) = 0$
  - This method has quadratic convergence (pretty fast!) provided we start close enough to the minimum
  - If derivatives are expensive to compute, we can fit a quadratic to three points to make the approximation, e.g. $f(x_{k-2}), f(x_{k-1}), f(x_k)$

Definition

A function $f : [a, b] \mapsto \mathbb{R}$ is *unimodular* if there exists $x^* \in [a, b]$ such that f is decreasing/nonincreasing for $x \in [a, x^*]$ and increasing/nondecreasing for $x \in [x^*, b]$, i.e. $f'(x) \leq 0$ for $x < x^*$ and $f'(x) \geq 0$ for $x > x^*$.

- What if our function is non-differentiable? We can exploit other properties of the function such as unimodularity
  - Suppose we evaluate f at x_0 and x_1; then if $f(x_0) > f(x_1)$ we know x_0 is not a minimum, so discard everything to the left; or if $f(x_0) < f(x_1)$ we know x_1 is not a minimum, so we discard everything to the right
  - Note that we can't discard both sides because that could potentially discard the section that the minimum is in
- This algorithm is called *golden section search*:
  - Steps:
    1. Without loss of generality, assume $a = 0, b = 1$ (rescale the function)
    2. Choose $x_0 = \alpha, x_1 = 1 - \alpha$ for $\alpha \in \left(0, \frac{1}{2}\right)$
    3. Discard one side of the interval based on $f(x_0)$ and $f(x_1)$, then rescale the interval and repeat
  - This algorithm has unconditional linear convergence as long as f is unimodular
  - We can further optimize the algorithm by choosing x_1 at the next iteration to be x_0 from the previous iteration to save one function evaluation, so $\alpha = (1 - \alpha)^2 \implies 1 - \alpha = \frac{1}{2}(\sqrt{5} - 1)$
    * This gives us the golden ratio, hence the name
- If f is neither unimodular nor differentiable, we need to exploit some other structure of f

Multidimensional Methods

- Suppose we want to minimize $f(\boldsymbol{x})$ where $f : \mathbb{R}^n \mapsto \mathbb{R}$; assume $f(\boldsymbol{x})$ is twice-differentiable
- Gradient descent requires only the first derivative, but is relatively slow (linear)

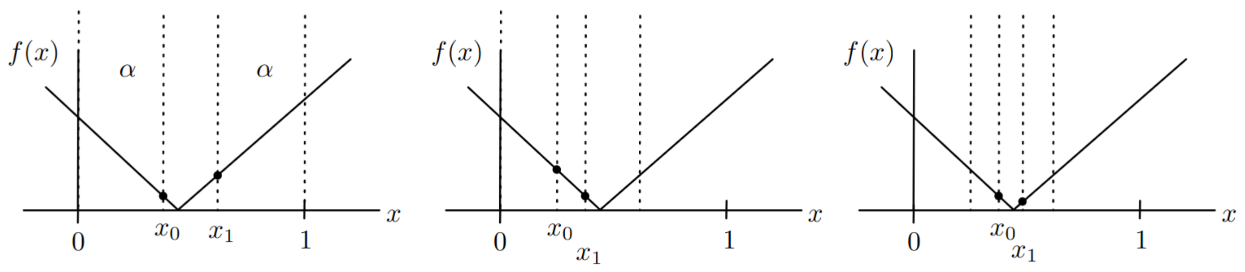

Figure 1: Illustration of golden section search.

- The idea is to iteratively take small steps in the local direction of steepest descent, which is in the opposite of direction as the gradient
- For a sufficiently small α, $f(\boldsymbol{x} - \alpha\vec{\nabla} f(\boldsymbol{x})^T) \leq f(\boldsymbol{x})$
- Steps:
  1. Let $g(\alpha) = f(\boldsymbol{x}_k - \alpha\vec{\nabla} f(\boldsymbol{x}_k)^T)$
  2. Find $\alpha^* = \min\limits_{\alpha} g(\alpha)$ through a one-dimensional line search on $g(\alpha)$ for $\alpha \geq 0$
  3. Update the estimate as $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha^*\vec{\nabla} f(\boldsymbol{x}_k)^T$
  4. Repeat until $\boldsymbol{x}_k$ changes sufficiently slowly
- Note that this allows us to convert a multidimensional optimization problem down to a single-variable optimization
- The linear search for α^* is so that we can get to the minimum faster and don't get stuck/jump around the minimum
- In practice the line search is expensive, so suboptimal techniques such as fixed α and backtracking are used, but these have no convergence guarantees
- Gradient descent is impacted by poor conditioning of $f(\boldsymbol{x})$; i.e. if $f(\boldsymbol{x})$ is changing rapidly in one dimension and slowly in another, it takes many iterations as the faster dimension is prioritized

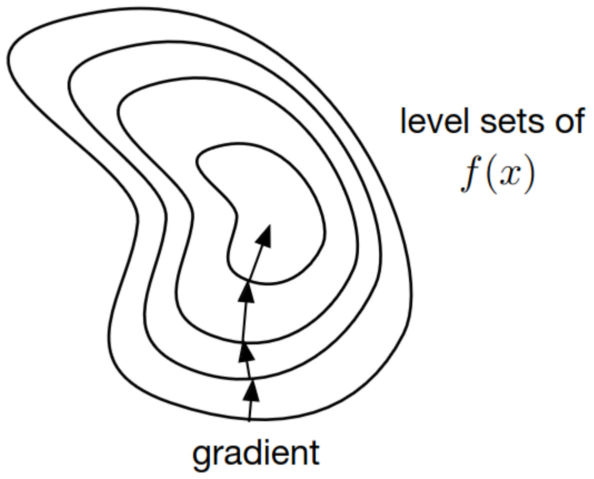

Figure 2: Illustration of gradient descent.

- Newton's method can be adapted for multiple dimensions by replacing the derivative by the gradient and the second derivative by the Hessian
  - $f(\boldsymbol{x}) \approx f(\boldsymbol{x}_k) + \vec{\nabla} f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)^T \boldsymbol{H}_f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k)$
  - Solving for $\vec{\nabla} f(\boldsymbol{x}) = \boldsymbol{0}$ for critical points gives $\boldsymbol{H}_f(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = -\vec{\nabla} f(\boldsymbol{x}_k)$
  - Therefore we can iterate as $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \boldsymbol{H}_f^{-1}(\boldsymbol{x}_k)\vec{\nabla} f(\boldsymbol{x}_k)$ and repeat until the change in $\boldsymbol{x}$ becomes small

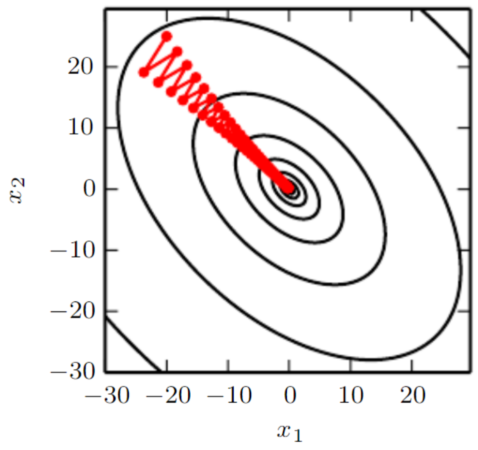

Figure 3: Gradient descent with poor conditioning.

- This helps with poor conditioning because applying $\boldsymbol{H}_f^{-1}$ effectively unwarps the problem
- The tradeoff is that Newton's method requires a Hessian to be inverted, which makes it slower and less applicable; there can also be issues with singularity of the Hessian
- This has quadratic convergence

- Common variants of Newton's method:
  - The Gauss-Newton method approximates the Hessian using first derivatives instead, which is a common compromise between gradient descent and Newton's method
  - Levenberg-Marquardt provides adaptive regularization to the Hessian when it is close to singular (effectively downgrading to gradient descent)

Note

In this course gradients are assumed to be row vectors, $\vec{\nabla} f(\boldsymbol{x}) \in \mathbb{R}^{1\times n}$ for $x \in \mathbb{R}^n$.

Example: Nonlinear Least Squares

- Let $e_i(\boldsymbol{\theta}) = y_i - g(x_i, \boldsymbol{\theta})$; we want to find $\boldsymbol{\theta}^*$ such that $e_i(\boldsymbol{\theta}^*)$ is minimized
  - This is applicable in e.g. neural network training (where θ are the weights), or system identification (where θ are the system parameters)
- The nonlinear least squares problem is $\min\limits_{\boldsymbol{\theta}} \sum\limits_i \|e_i(\boldsymbol{\theta})\|^2$, or equivalently $\min\limits_{\boldsymbol{\theta}} \boldsymbol{e}(\boldsymbol{\theta})^T\boldsymbol{e}(\boldsymbol{\theta})$
- Gradient: $\vec{\nabla} f(\boldsymbol{\theta}_k) = \dfrac{\partial f}{\partial \boldsymbol{e}}\dfrac{\partial \boldsymbol{e}}{\partial \boldsymbol{\theta}} = 2\boldsymbol{e}(\theta)^T\dfrac{\partial \boldsymbol{e}(\theta)}{\partial \boldsymbol{\theta}}$ where $\dfrac{\partial \boldsymbol{e}(\theta)}{\partial \boldsymbol{\theta}} = \boldsymbol{J}$ is the Jacobian
- Hessian: $\boldsymbol{H}_f = \vec{\nabla}^2 f(\boldsymbol{\theta}_k) = 2\left(\dfrac{\partial \boldsymbol{e}(t)}{\partial \boldsymbol{\theta}}^T\dfrac{\partial \boldsymbol{e}(t)}{\partial \boldsymbol{\theta}} + \boldsymbol{e}(\boldsymbol{\theta})^T\vec{\nabla}^2_{\boldsymbol{\theta}}\boldsymbol{e}(\boldsymbol{\theta})\right)$
  - Note we get $\dfrac{\partial \boldsymbol{e}(t)}{\partial \boldsymbol{\theta}}^T\dfrac{\partial \boldsymbol{e}(t)}{\partial \boldsymbol{\theta}} = \boldsymbol{J}^T\boldsymbol{J}$ for free from the gradient, but the second term $\boldsymbol{e}(\boldsymbol{\theta})^T\vec{\nabla}^2_{\boldsymbol{\theta}}\boldsymbol{e}(\boldsymbol{\theta})$ is expensive
  - Since the second term involves the error, we can approximate it as zero assuming that we are close to the optimum
- Using Newton's method, $\boldsymbol{H}_f\Delta\boldsymbol{\theta} = 2\boldsymbol{e}(\boldsymbol{\theta}_k)^T\boldsymbol{J}(\boldsymbol{\theta}_k) \implies \Delta\boldsymbol{\theta} = (\boldsymbol{J}\boldsymbol{J}^T)^{-1}\boldsymbol{J}^T\boldsymbol{e}(\boldsymbol{\theta}_k)$
- Making this Hessian approximation is known as the Gauss-Newton method
- In practice we use techniques to make sure $\boldsymbol{J}\boldsymbol{J}^T$ remains sparse, so it can be inverted quickly

- Most SLAM algorithms use some form of Gauss-Newton