Unconstrained optimization

Nick Henderson, AJ Friend (Stanford University)
Kevin Carlberg (Sandia National Laboratories)

August 13, 2019

Unconstrained optimization

Theory, methods, and software for problems exihibiting the characteristics below

- Convexity:
  - convex: local solutions are global
  - $\boxed{\text{non-convex}}$: local solutions are not global
- Optimization-variable type:
  - $\boxed{\text{continuous}}$: gradients facilitate computing the solution
  - discrete: cannot compute gradients, NP-hard
- Constraints:
  - $\boxed{\text{unconstrained}}$: simpler algorithms
  - constrained: more complex algorithms; must consider feasibility
- Number of optimization variables:
  - $\boxed{\text{low-dimensional}}$: can solve even without gradients
  - $\boxed{\text{high-dimensional}}$: requires gradients to be solvable in practice

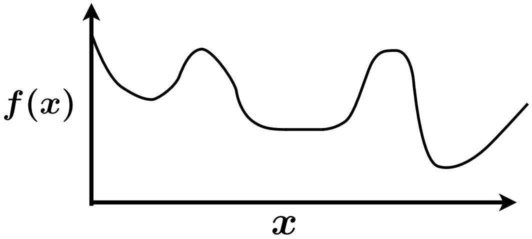
Theory

Outline

Unconstrained optimization in one variable

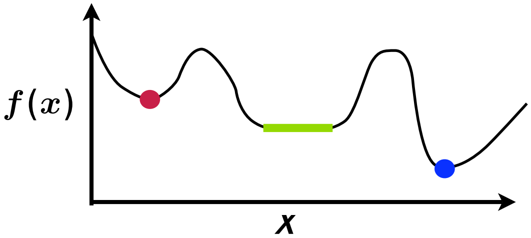
$$\text{minimize} \quad f(x)$$

- $x \in \mathbf{R}$ is a real-valued variable
- $f(x) \in C^2 : \mathbf{R} \to \mathbf{R}$ is the objective function, which returns a single real number



- What is a solution to this problem?

What is a solution?

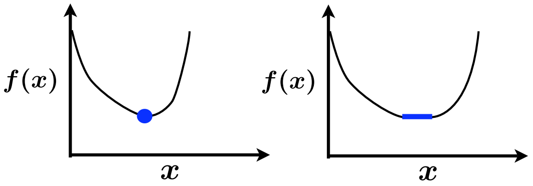


- Global minumum: A point $x^\star$ satisfying $f(x^\star) \leq f(x)$ for all x in the domain of interest
- Strong local minumum: A point $x^\star$ satisfying $f(x^\star) < f(x)$ for all x in a neighborhood of $x^\star$
- Weak local minumum: A point $x^\star$ satisfying $f(x^\star) \leq f(x)$ for all x in a neighborhood of $x^\star$

Convexity

- For a convex objective function in one variable,

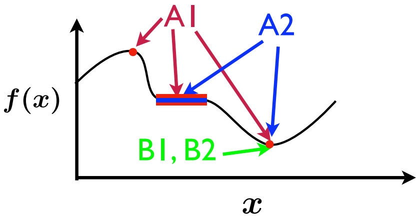
$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y)$$



- Any local minimum is a global minimum!

Optimality conditions for single-variable minimization

- Necessary conditions for a weak local minimum:
  - **A1**: $f'(x^\star) = 0$
  - **A2**: $f''(x^\star) \geq 0$
- Sufficient conditions for a strong local minimum:
  - **B1**: $f'(x^\star) = 0$, and
  - **B2**: $f''(x^\star) > 0$
- **Stationary point**: a point $x^\star$ satisfying $f'(x^\star) = 0$
- **Saddle point**: a stationary point that is not a local minimum or maximum



The gradient and Hessian are absolutely essential quantities

Unconstrained optimization in multiple variables

$$\text{minimize} \quad f(x)$$

- $x \in \mathbf{R}^n$ is an n-dimensional vector of real numbers
- $f(x) \in C^2 : \mathbf{R}^n \to \mathbf{R}$ is the objective function, which returns a single real number
- The same notions of weak local, strong local, and global minima, as well as convexity, extend to multiple dimensions

Derivatives in multiple dimensions

Vector of optimization variables:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Gradient (i.e., first derivative) of f:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

Hessian (i.e., second derivative) of f:

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix}$$

Stationary points

- **Stationary point**: a point $x^\star$ satisfying $\nabla f(x^\star) = 0$

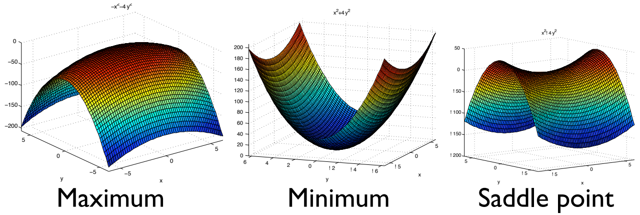


Figure 1: Types of stationary points in two dimensions

Optimality conditions for multiple-variable minimization

Can simply extend the univariate conditions to multiple dimensions

- Necessary conditions for a weak local minimum:
  - **A1**: $\nabla f(x^\star) = 0$
  - **A2**: $\nabla^2 f(x^\star) \succeq 0$
  - $\nabla^2 f(x) \succeq 0$ means that all the eigenvalues of $\nabla^2 f(x)$ are non-negative
- Sufficient conditions for a strong local minimum:
  - **B1**: $\nabla f(x^\star) = 0$, and
  - **B2**: $\nabla^2 f(x^\star) \succ 0$
  - $\nabla^2 f(x) \succ 0$ means that all the eigenvalues of $\nabla^2 f(x)$ are strictly positive

Algorithms

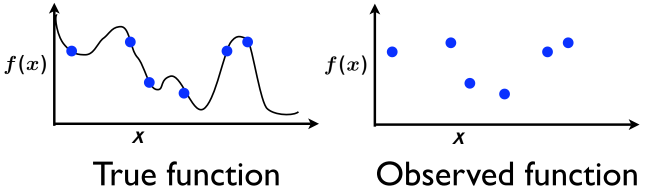
Outline

Optimization algorithms

- We now know:
  - What an unconstrained optimization problem is
  - How to characterize local/global solutions using optimality conditions
- How do we *compute* these solutions?
  - **Analytically**: only possible for very simple problems (e.g., Brachistochrone problem)
  - **Numerically**: required for most practical problems
- **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 may be computationally expensive



- Derivative information (gradient ∇f and Hessian $\nabla^2 f$) is sometimes available
  - generally more expensive to compute
  - can help *a lot* (determine optimality criteria)
  - especially helpful in high dimensions (n large)

Optimization algoritihms

- Goals
  - **Practical**: reasonable memory requirements
  - **Robust**: low failure rate, convergence conditions are met
  - **Fast**: convergence in a few iterations, low cost per iteration
  - Application typically dictates specific requirements
- Algorithm design involves tradeoffs to achieve these goals
  - *Example*: using derivatives may reduce number of iterations, but each iteration becomes more expensive
- Algorithms are **iterative** in nature
- Categorization
  - **Gradient-based** v. **derivative-free**
  - **Global** v. **local**: aims to converge to a global or local minimum
  - Gradient-based algorithms tend to be local, while derivative-free algorithms tend to be global

Gradient-based algorithms

Outline

Gradient-based algorithms

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

- How would you get down (i.e., find the minimum)?
- 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

Gradient-based algorithms: benefits and drawbacks

- **Benefits**
  - Efficient for many variables (i.e., in high dimensions)
  - Well-suited for smooth objective and constraint functions
- **Drawbacks**
  - Requires computing the gradient (challenging in some cases)
  - Convergence is only local (local optimization)
    - Mitigated by using multiple initial guesses to find multiple local minima
    - Can then choose the best local minimum
  - Not well-suited for discrete optimization
  - Not well-suited for noisy functions
- Second derivatives (Hessians) are also very valuable
  - However, Hessians are $n \times n$ symmetric matrices, so expensive to construct and store
  - If Hessians are needed, they are often approximated using **quasi-Newton methods**

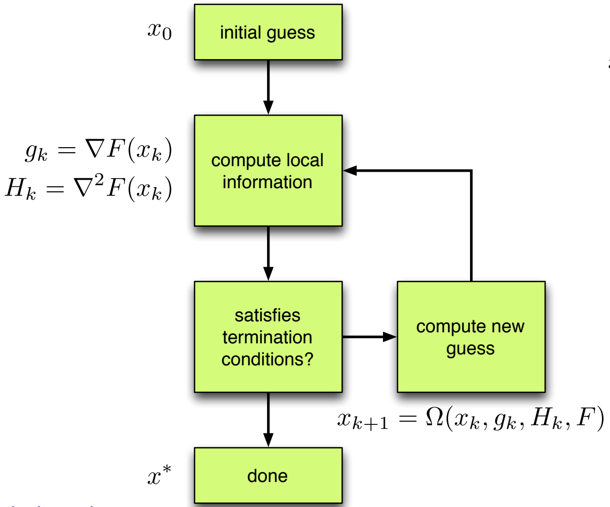
Gradient-based algorithms: framework

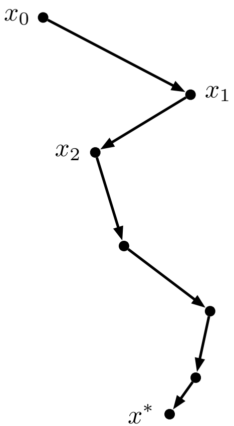
- At each iteration k, gradient-based methods compute both
  1. a **search direction** p_k, and
  2. a **step length** α_k (referred to as the **learning rate** in machine learning)

Algorithm 1 Gradient-based framework

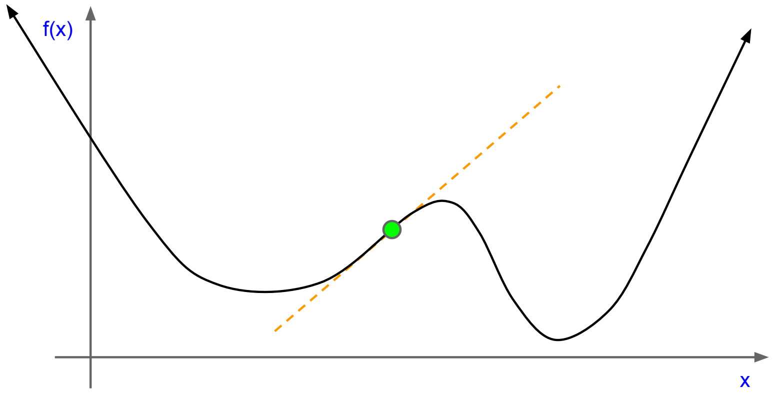
```
Choose initial guess $x_0$, set $k \leftarrow 0$
while (not converged) do
    $\boxed{\text{Choose direction } p_k \text{ and step length } \alpha_k}$
        (This often involves computing local information, e.g., $\nabla f(x_k)$, $\nabla^2 f(x_k)$)
    $x_{k+1} = x_k + \alpha_k p_k$.
    $k \leftarrow k + 1$
end while
```

Gradient-based algorithms: overview

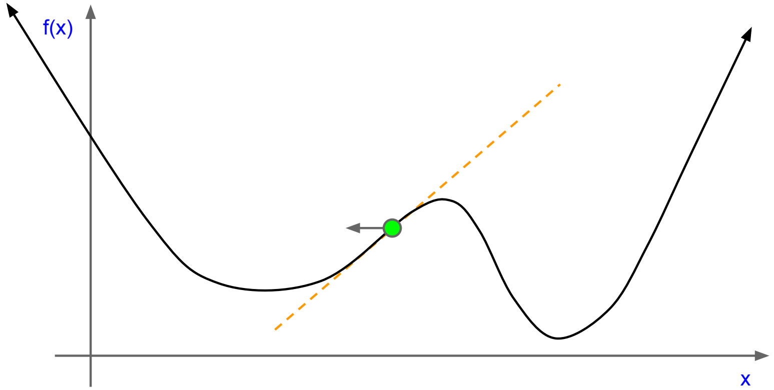

x_0
initial guess
$g_k = \nabla F(x_k)$
$H_k = \nabla^2 F(x_k)$
compute local information
satisfies termination conditions?
compute new guess
$x_{k+1} = \Omega(x_k, g_k, H_k, F)$
x^*
done



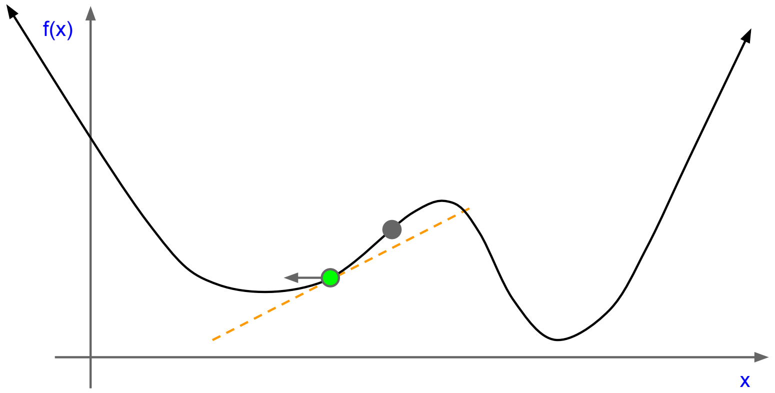
Gradient-based algorithms: sketch



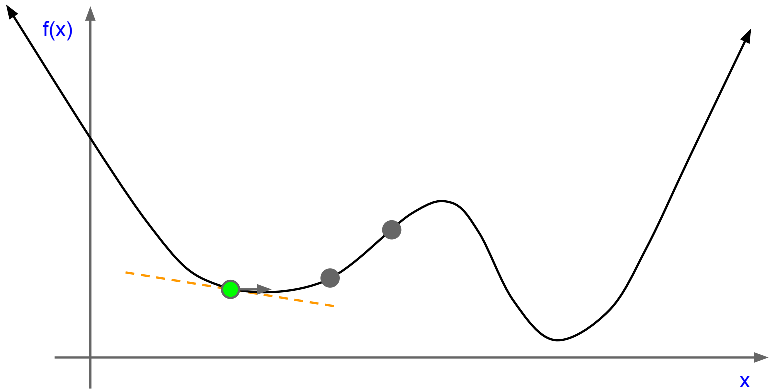
Gradient-based algorithms: sketch



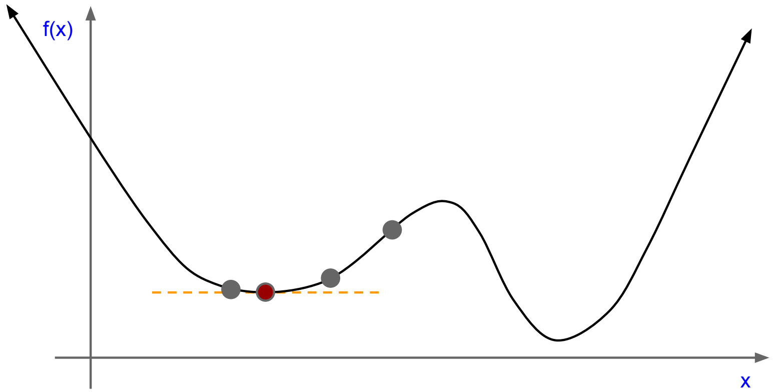
Gradient-based algorithms: sketch



Gradient-based algorithms: sketch



Gradient-based algorithms: sketch



Gradient-based algorithms: two classes

There are two classes of gradient-based algorithms.

- **Line-search methods**:
  1. compute p_k to be a descent direction
  2. compute α_k to produce a sufficient decrease in the objective function
- **Trust-region methods**:
  1. determine a maximum allowable step length (trust-region radius) Δ_k,
  2. compute step p_k with $\|p_k\| \leq \Delta$ using a model $m(p) \approx f(x_k + p)$
  3. **accept step** if actual objective-function reduction is close to (or better than) the model-preducted objective-function reduction, and set $x_{k+1} = x_k + p_k$ (note $\alpha_k = 1$)
  4. otherwise, **reject step**, set $x_{k+1} = x_k$, and shrink trust-region radius such that $\Delta_{k+1} < \Delta_k$

Gradient-based algorithms: two classes

There are two classes of gradient-based algorithms.

- $\boxed{\textbf{Line-search methods}}$:
  1. compute p_k to be a descent direction
  2. compute α_k to produce a sufficient decrease in the objective function
- **Trust-region methods**:
  1. determine a maximum allowable step length (trust-region radius) Δ_k,
  2. compute step p_k with $\|p_k\| \leq \Delta$ using a model $m(p) \approx f(x_k + p)$
  3. **accept step** if actual objective-function reduction is close to (or better than) the model-preducted objective-function reduction, and set $x_{k+1} = x_k + p_k$ (note $\alpha_k = 1$)
  4. otherwise, **reject step**, set $x_{k+1} = x_k$, and shrink trust-region radius such that $\Delta_{k+1} < \Delta_k$

Line-search methods: convergence

Theorem (Sufficient conditions for convergence)

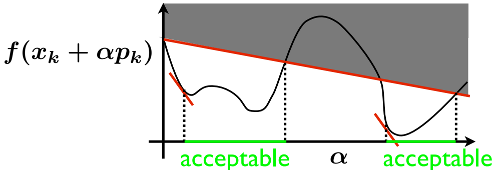
For sufficiently smooth, well-defined problems, sufficient conditions for convergence $\lim_{k\to\infty} \|\nabla f_k\| = 0$ *of line search methods are:*

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

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

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

- **Decrease** f: $f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T p_k$
- **Increase** ∇f: $\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k$.



Line-search methods: key steps

1. Choose search direction p_k that is a descent direction (satisfy C1)
2. Choose step length α_k that satsifies the Wolfe conditions (satisfy C2)

Line-search methods: step 1 (gradient descent)

Choose search direction p_k that is a descent direction (satisfy C1)

- **Gradient descent** (i.e., steepest descent): $p_k = -\nabla f(x_k)$
  - Steepest direction downhill
  - **Advantages**: only first-order information, always a descent direction, low storage
  - **Disadvantages**: linear convergence rate, sensitive to variable scaling
  - **Stochastic gradient descent** is an *approximation* of this yielding sublinear convergence
- **Conjugate gradient**: $p_k = -\nabla f(x_k) + \beta_k p_{k-1}$
  - β_k computed to ensure p_k and p_{k-1} are approximately conjugate (accounts for previous progress)
  - Linear (faster) convergence
  - **Advantages**: only first-order information, low storage, more effective than steepest descent and almost as simple to implement
  - **Disadvantages**: linear convergence rate (but faster than steepest descent), sensitive to variable scaling

Line-search methods: step 1 (modified Newton's method)

Choose search direction p_k that is a descent direction (satisfy C1)

Recall that $\nabla f(x^\star) = 0$ is a necessary condition for optimality

- This is just n nonlinear equations in n unknowns!
- Thus, we could apply **Newton's method** to solve it and obtain quadratic convergence!
- This would lead to $p_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$
- If f is strongly convex quadratic and $\alpha_k = 1$, this converges in **one iteration**!
    - *Scale invariant*: this holds regardless of variable scaling
    - *Natural step length*: $\alpha_k = 1$
    - The Hessian overcomes issues with ill-conditioning/poorly scaled variables
- However, p_k is not guaranteed to be a descent direction (i.e., might not satisfy C1)
- So, we must *modify* Newton's method to ensure p_k is a descent direction

Line-search methods: step 1 (modified Newton's method)

- **Modified Newton's method**: $p_k = -(\nabla^2 f(x_k) + E_k)^{-1} \nabla f(x_k)$
  - if $\nabla^2 f(x_k) + E_k$ is positive definite, then p_k is a descent direction
  - Thus, E_k is computed to ensure $\nabla^2 f(x_k) + E_k$ is positive definite
  - **Advantages**: quadratic convergence, scale invariant, natural step length
  - **Disadvantage**: second-order information (expensive), large storage
- **Quasi-Newton methods**: $p_k = -(B_k)^{-1} \nabla f(x_k)$
  - B_k updated each iteration using the only the gradient to satisfy the *secant condition*

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

  - Popular updates:
    - *Symmetric rank-one (SR1)*: enforces symmetry, rank 1
    - *Broyden, Fletcher, Goldfarb, Shanno (BFGS)*: enforces positive definiteness, rank 2
  - **Advantages**: only first-order information, superlinear convergence, scale invariant, natural step length, limited-memory variant L-BFGS ensures low storage
  - **Disadvantages**: may not be a descent direction (e.g., if SR1), approximate Hessians may be inaccurate and dense

Line-search methods: step 2

Choose step length α_k that satisfies the Wolfe conditions (satisfy C2)

- **Backtracking**:
  - **Goal**: given p_k find α such that $f(x_k + \alpha p_k) < f(x_k)$.
  - **Procedure**: start with initial guess $\alpha > 0$ (use $\alpha = 1$ for Newton's method)
  1. if $f(x_k + \alpha p_k) < f(x_k)$, then return α, otherwise continue
  2. decrease α by some factor $0 < \delta < 1$: $\alpha \leftarrow \delta\alpha$
  3. repeat

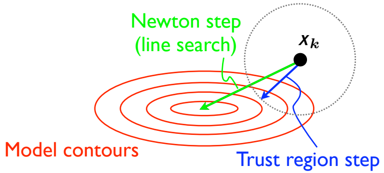
Gradient-based algorithms: two classes

There are two classes of gradient-based algorithms.

- **Line-search methods**:
  1. compute p_k to be a descent direction
  2. compute α_k to produce a sufficient decrease in the objective function
- $\boxed{\textbf{Trust-region methods}}$:
  1. determine a maximum allowable step length (trust-region radius) Δ_k,
  2. compute step p_k with $\|p_k\| \leq \Delta$ using a model $m(p) \approx f(x_k + p)$
  3. **accept step** if actual objective-function reduction is close to (or better than) the model-preducted objective-function reduction, and set $x_{k+1} = x_k + p_k$ (note $\alpha_k = 1$)
  4. otherwise, **reject step**, set $x_{k+1} = x_k$, and shrink trust-region radius such that $\Delta_{k+1} < \Delta_k$

Trust-region methods: overview

- Trust region methods sequentially minimize an *approximate, easy-to-solve* model within a local trust region
- The trust region is the region within which the approximate model is *trusted*
- The subproblem is often *convex* (**sequential convex programming**)



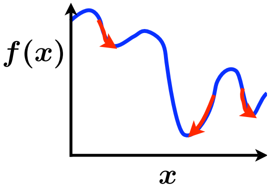
- If the step is unacceptable (inaccurate model), the size of the trust region is reduced (we trust the model less) and minimization is repeated around the same point

Trust region methods

- Trust region methods often 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
$$\text{minimize } m_k(p) \quad \text{subject to } \|p\| \leq \Delta_k$$

Gradient-based algorithms for global optimization

- Gradient-based algorithms are best-suited for finding local minima: they "go downhill" until local optimality conditions are satisfied
- To find multiple local minima (and hopefully the global minimum), gradient-based methods can be run multiple times using different initial guesses



- However even if we happen to find the global minimum, we cannot verify that we have done so!
- This tuning/babysitting does not arise in convex optimization!

Computation of gradients

- To implement gradient-based algorithms, derivative information must be computed
- There are three main ways to compute these gradients
  1. Analytical (can use symbolic tools, e.g., Mathematica)
  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

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

- Tradeoff:
    - ϵ too large: inaccurate due to truncation error
    - ϵ too small: inaccurate due to subtractive cancellation from round-off error

Automatic differentiation

- Use computational representation of a function
- Key observations:
  - Any function is composed of a sequence of simple operations
  - 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
- **Backpropagation** in deep learning is a specific case of automatic differentiation

Derivative-free algorithms

Outline

Why derivative-free algorithms?

- Gradients may not be available
  - $f(x)$ from laboratory experiments
  - impractical or cumbersome to implement analytic gradinets
- Noise or non-smoothness in the objective function
  - this creates *many local minima*, so gradient information less useful
  - require global optimization
- May want to direct effort *globally* (less information at more points) rather than *locally* (more information at fewer points)
- Can use global optimization to define initial guesses for local optimization

Benefits and drawbacks of derivative-free algorithms

- **Benefits**:
  - Well-suited for discrete variables
  - Often better at finding the global optimum (if non-convex)
  - Robust with respect to noise in the function
  - Useful for multi-objective optimization
  - Amenable to parallelization
- **Drawbacks**:
  - **Extremely slow convergence in high dimensions (n large)**
  - Difficult to efficiently treat constraints
- Not typically used if gradients are available

Derivative-free algorithm categorization

- **Heuristic**: use techniques inspired by nature (global optimization)
  - Simulated annealing
  - Basin hopping (Monte Carlo)
  - Evolutionary techniques
    - Genetic algorithms
    - Differential evolution
    - Swarm intelligence (particle swarm optimization, ant colony optimization)
- **Direct search**: query a sequence of nearby points (local optimization)
  - *Directional*: coordinate search (e.g., Powell's method), pattern search
  - *Simplicial*: Nelder–Mead nonlinear simplex

Evolutionary Algorithms

- Evolutionary algorithms 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** (recombination and mutation) and **selection** (survival of the fittest)

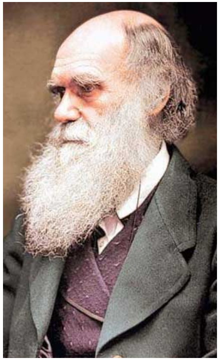


Figure 2: Charles Darwin

Evolutionary Algorithm

$$\text{minimize} \quad f(x)$$

- A population member is represented by a point x in the variable space (its DNA)
- 'Fitness' is the objective function value $f(x)$
- At each iteration, rather than work with a *single point*, we consider an entire *population of points* across the entire space
- **Benefit**: more likely to find a global optimum and won't be "trapped" by local minima
- **Drawback**: very expensive in high dimensions

Overview of evolutionary 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