AM 205: lecture 20

- Last time: Broyden's method / BFGS
- Today: Conjugate gradient, linear programming, adjoint-based PDE optimization

Conjugate Gradient Method

The conjugate gradient (CG) method is another alternative to Newton's method that does not require the Hessian:

```
1: choose initial guess $x_0$
2: $g_0 = \nabla f(x_0)$
3: $x_0 = -g_0$
4: for $k = 0, 1, 2, \ldots$ do
5:     choose $\eta_k$ to minimize $f(x_k + \eta_k s_k)$
6:     $x_{k+1} = x_k + \eta_k s_k$
7:     $g_{k+1} = \nabla f(x_{k+1})$
8:     $\beta_{k+1} = (g_{k+1}^T g_{k+1})/(g_k^T g_k)$
9:     $s_{k+1} = -g_{k+1} + \beta_{k+1} s_k$
10: end for
```

Constrained Optimization

Equality Constrained Optimization

We now consider equality constrained minimization:

$$\min_{x \in \mathbb{R}^n} f(x) \quad \text{subject to} \quad g(x) = 0,$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$

With the Lagrangian $\mathcal{L}(x, \lambda) = f(x) + \lambda^T g(x)$, we recall from that necessary condition for optimality is

$$\nabla \mathcal{L}(x, \lambda) = \begin{bmatrix} \nabla f(x) + J_g^T(x)\lambda \\ g(x) \end{bmatrix} = 0$$

Once again, this is a nonlinear system of equations that can be solved via Newton's method

Sequential Quadratic Programming

To derive the Jacobian of this system, we write

$$\nabla \mathcal{L}(x, \lambda) = \left[\begin{array}{c} \nabla f(x) + \sum_{k=1}^{m} \lambda_k \nabla g_k(x) \\ g(x) \end{array} \right] \in \mathbb{R}^{n+m}$$

Then we need to differentiate wrt to $x \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}^m$

For $i = 1, \ldots, n$, we have

$$(\nabla \mathcal{L}(x, \lambda))_i = \frac{\partial f(x)}{\partial x_i} + \sum_{k=1}^{m} \lambda_k \frac{\partial g_k(x)}{\partial x_i}$$

Differentiating wrt x_j, for $i, j = 1, \ldots, n$, gives

$$\frac{\partial}{\partial x_j} (\nabla \mathcal{L}(x, \lambda))_i = \frac{\partial^2 f(x)}{\partial x_i \partial x_j} + \sum_{k=1}^{m} \lambda_k \frac{\partial^2 g_k(x)}{\partial x_i \partial x_j}$$

Sequential Quadratic Programming

Hence the top-left $n \times n$ block of the Jacobian of $\nabla \mathcal{L}(x, \lambda)$ is

$$B(x, \lambda) \equiv H_f(x) + \sum_{k=1}^{m} \lambda_k H_{g_k}(x) \in \mathbb{R}^{n \times n}$$

Differentiating $(\nabla \mathcal{L}(x, \lambda))_i$ wrt λ_j, for $i = 1, \ldots, n$, $j = 1, \ldots, m$, gives

$$\frac{\partial}{\partial \lambda_j} (\nabla \mathcal{L}(x, \lambda))_i = \frac{\partial g_j(x)}{\partial x_i}$$

Hence the top-right $n \times m$ block of the Jacobian of $\nabla \mathcal{L}(x, \lambda)$ is

$$J_g(x)^T \in \mathbb{R}^{n \times m}$$

Sequential Quadratic Programming

For $i = n+1, \dots, n+m$, we have

$$(\nabla \mathcal{L}(x, \lambda))_i = g_i(x)$$

Differentiating $(\nabla \mathcal{L}(x, \lambda))_i$ wrt x_j, for $i = n+1, \dots, n+m$, $j = 1, \dots, n$, gives

$$\frac{\partial}{\partial x_j}(\nabla \mathcal{L}(x, \lambda))_i = \frac{\partial g_i(x)}{\partial x_j}$$

Hence the bottom-left $m \times n$ block of the Jacobian of $\nabla \mathcal{L}(x, \lambda)$ is

$$J_g(x) \in \mathbb{R}^{m \times n}$$

$\dots$ and the final $m \times m$ bottom right block is just zero (differentiation of $g_i(x)$ w.r.t. λ_j)

Sequential Quadratic Programming

Hence, we have derived the following Jacobian matrix for $\nabla \mathcal{L}(x, \lambda)$:

$$\begin{bmatrix} B(x, \lambda) & J_g^T(x) \\ J_g(x) & 0 \end{bmatrix} \in \mathbb{R}^{(m+n)\times(m+n)}$$

Note the 2×2 block structure of this matrix (matrices with this structure are often called KKT matrices[1])

[1] Karush, Kuhn, Tucker: did seminal work on nonlinear optimization

Sequential Quadratic Programming

Therefore, Newton's method for $\nabla \mathcal{L}(x, \lambda) = 0$ is:

$$\begin{bmatrix} B(x_k, \lambda_k) & J_g^T(x_k) \\ J_g(x_k) & 0 \end{bmatrix} \begin{bmatrix} s_k \\ \delta_k \end{bmatrix} = - \begin{bmatrix} \nabla f(x_k) + J_g^T(x_k)\lambda_k \\ g(x_k) \end{bmatrix}$$

for $k = 0, 1, 2, \dots$

Here $(s_k, \delta_k) \in \mathbb{R}^{n+m}$ is the k^{th} Newton step

Sequential Quadratic Programming

Now, consider the constrained minimization problem, where (x_k, λ_k) is our Newton iterate at step k:

$$\min_s \left\{ \frac{1}{2} s^T B(x_k, \lambda_k) s + s^T (\nabla f(x_k) + J_g^T(x_k)\lambda_k) \right\}$$

$$\text{subject to} \quad J_g(x_k)s + g(x_k) = 0$$

The objective function is quadratic in s (here x_k, λ_k are constants)

This minimization problem has Lagrangian

$$\begin{aligned} \mathcal{L}_k(s, \delta) \ &\equiv\ \frac{1}{2} s^T B(x_k, \lambda_k) s + s^T (\nabla f(x_k) + J_g^T(x_k)\lambda_k) \\ &+\ \delta^T (J_g(x_k)s + g(x_k)) \end{aligned}$$

Sequential Quadratic Programming

Then solving $\nabla \mathcal{L}_k(s, \delta) = 0$ (i.e. first-order necessary conditions) gives a linear system, which is the same as the kth Newton step

Hence at each step of Newton's method, we exactly solve a minimization problem (quadratic objective fn., linear constraints)

An optimization problem of this type is called a quadratic program

This motivates the name for applying Newton's method to $\mathcal{L}(x, \lambda) = 0$: Sequential Quadratic Programming (SQP)

Sequential Quadratic Programming

SQP is an important method, and there are many issues to be considered to obtain an efficient and reliable implementation:

- Efficient solution of the linear systems at each Newton iteration — matrix block structure can be exploited
- Quasi-Newton approximations to the Hessian (as in the unconstrained case)
- Trust region, line search etc to improve robustness
- Treatment of constraints (equality and inequality) during the iterative process
- Selection of good starting guess for λ

Penalty Methods

Another computational strategy for constrained optimization is to employ penalty methods

This converts a constrained problem into an unconstrained problem

Key idea: Introduce a new objective function which is a weighted sum of objective function and constraint

Penalty Methods

Given the minimization problem

$$\min_x f(x) \quad \text{subject to} \quad g(x) = 0$$

we can consider the related unconstrained problem

$$\min_x \phi_\rho(x) = f(x) + \frac{1}{2}\rho g(x)^T g(x) \qquad (**)$$

Let x^* and x^*_ρ denote the solution of $(*)$ and $(**)$, respectively

Under appropriate conditions, it can be shown that

$$\lim_{\rho \to \infty} x^*_\rho = x^*$$

Penalty Methods

In practice, we can solve the unconstrained problem for a large value of ρ to get a good approximation of x^*

Another strategy is to solve for a sequence of penalty parameters, ρ_k, where $x^*_{\rho_k}$ serves as a starting guess for $x^*_{\rho_{k+1}}$

Note that the major drawback of penalty methods is that a large factor ρ will increase the condition number of the Hessian H_{ϕ_ρ}

On the other hand, penalty methods can be convenient, primarily due to their simplicity

Linear Programming

Linear Programming

As we mentioned earlier, the optimization problem

$$\min_{x \in \mathbb{R}^n} f(x) \text{ subject to } g(x) = 0 \text{ and } h(x) \leq 0, \qquad (*)$$

with f, g, h affine, is called a linear program

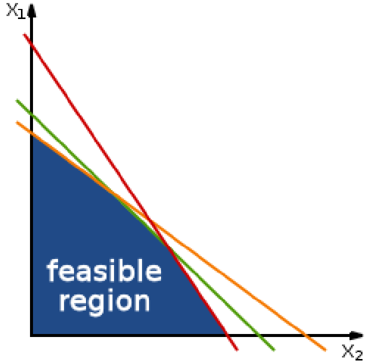
The feasible region is a convex polyhedron[2]

Since the objective function maps out a hyperplane, its global minimum must occur at a vertex of the feasible region

[2]Polyhedron: a solid with flat sides, straight edges

Linear Programming

This can be seen most easily with a picture (in $\mathbb{R}^2$)



Linear Programming

The standard approach for solving linear programs is conceptually simple: examine a sequence of the vertices to find the minimum

This is called the simplex method

Despite its conceptual simplicity, it is non-trivial to develop an efficient implementation of this algorithm

We will not discuss the implementation details of the simplex method...

Linear Programming

In the worst case, the computational work required for the simplex method grows exponentially with the size of the problem

But this worst-case behavior is extremely rare; in practice simplex is very efficient (computational work typically grows linearly)

Newer methods, called interior point methods, have been developed that are polynomial in the worst case

Nevertheless, simplex is still the standard approach since it is more efficient than interior point for most problems

Linear Programming

Python example: Using `cvxopt`, solve the linear program

$$\min_{x} f(x) = -5x_1 - 4x_2 - 6x_3$$

subject to

$$\begin{aligned} x_1 - x_2 + x_3 &\leq 20 \\ 3x_1 + 2x_2 + 4x_3 &\leq 42 \\ 3x_1 + 2x_2 &\leq 30 \end{aligned}$$

and $0 \leq x_1, 0 \leq x_2, 0 \leq x_3$

(LP solvers are efficient, main challenge is to formulate an optimization problem as a linear program in the first place!)

PDE Constrained Optimization

PDE Constrained Optimization

We will now consider optimization based on a function that depends on the solution of a PDE

Let us denote a parameter dependent PDE as

$$\mathrm{PDE}(u(p); p) = 0$$

- $p \in \mathbb{R}^n$ is a parameter vector; could encode, for example, the flow speed and direction in a convection–diffusion problem
- $u(p)$ is the PDE solution for a given p

PDE Constrained Optimization

We then consider an output functional g,[3] which maps an arbitrary function v to $\mathbb{R}$

And we introduce a parameter dependent output, $\mathcal{G}(p) \in \mathbb{R}$, where $\mathcal{G}(p) \equiv g(u(p)) \in \mathbb{R}$, which we seek to minimize

At the end of the day, this gives a standard optimization problem:

$$\min_{p \in \mathbb{R}^n} \mathcal{G}(p)$$

[3]A functional is just a map from a vector space to $\mathbb{R}$

PDE Constrained Optimization

One could equivalently write this PDE-based optimization problem as

$$\min_{p,u} g(u) \quad \text{subject to} \quad \text{PDE}(u; p) = 0$$

For this reason, this type of optimization problem is typically referred to as PDE constrained optimization

- objective function g depends on u
- u and p are related by the PDE constraint

Based on this formulation, we could introduce Lagrange multipliers and proceed in the usual way for constrained optimization...

PDE Constrained Optimization

Here we will focus on the form we introduced first:

$$\min_{p \in \mathbb{R}^n} \mathcal{G}(p)$$

Optimization methods usually need some derivative information, such as using finite differences to approximate $\nabla \mathcal{G}(p)$

PDE Constrained Optimization

But using finite differences can be expensive, especially if we have many parameters:

$$\frac{\partial \mathcal{G}(p)}{\partial p_i} \approx \frac{\mathcal{G}(p + he_i) - \mathcal{G}(p)}{h},$$

hence we need $n + 1$ evaluations of $\mathcal{G}$ to approximate $\nabla \mathcal{G}(p)$!

We saw from the Himmelblau example that supplying the gradient $\nabla \mathcal{G}(p)$ cuts down on the number of function evaluations required

The extra function calls due to F.D. isn't a big deal for Himmelblau's function, each evaluation is very cheap

But in PDE constrained optimization, each $p \to \mathcal{G}(p)$ requires a full PDE solve!

PDE Constrained Optimization

Hence for PDE constrained optimization with many parameters, it is important to be able to compute the gradient more efficiently

There are two main approaches:

- the direct method
- the adjoint method

The direct method is simpler, but the adjoint method is much more efficient if we have many parameters

PDE Output Derivatives

Consider the ODE BVP

$$-u''(x;p) + r(p)u(x;p) = f(x), \qquad u(a) = u(b) = 0$$

which we will refer to as the primal equation

Here $p \in \mathbb{R}^n$ is the parameter vector, and $r : \mathbb{R}^n \to \mathbb{R}$

We define an output functional based on an integral

$$g(v) \equiv \int_a^b \sigma(x)u(x)\mathrm{d}x,$$

for some function σ; then $\mathcal{G}(p) \equiv g(u(p)) \in \mathbb{R}$

The Direct Method

We observe that

$$\frac{\partial \mathcal{G}(p)}{\partial p_i} = \int_a^b \sigma(x) \frac{\partial u}{\partial p_i} \mathrm{d}x$$

hence if we can compute $\frac{\partial u}{\partial p_i}$, $i = 1, 2, \ldots, n$, then we can obtain the gradient

Assuming sufficient smoothness, we can "differentiate the ODE BVP" wrt p_i to obtain,

$$-\frac{\partial u}{\partial p_i}''(x; p) + r(p) \frac{\partial u}{\partial p_i}(x; p) = -\frac{\partial r}{\partial p_i} u(x; p)$$

for $i = 1, 2, \ldots, n$

The Direct Method

Once we compute each $\frac{\partial u}{\partial p_i}$ we can then evaluate $\nabla \mathcal{G}(p)$ by evaluating a sequence of n integrals

However, this is not much better than using finite differences: We still need to solve n separate ODE BVPs

(Though only the right-hand side changes, so could LU factorize the system matrix once and back/forward sub. for each i)

Adjoint-Based Method

However, a more efficient approach when n is large is the adjoint method

We introduce the adjoint equation:

$$-z''(x;p) + r(p)z(x;p) = \sigma(x), \qquad z(a) = z(b) = 0$$

Adjoint-Based Method

Now,

$$\begin{aligned}
\frac{\partial \mathcal{G}(p)}{\partial p_i} &= \int_a^b \sigma(x) \frac{\partial u}{\partial p_i} \mathrm{d}x \\
&= \int_a^b \left[-z''(x;p) + r(p) z(x;p) \right] \frac{\partial u}{\partial p_i} \mathrm{d}x \\
&= \int_a^b z(x;p) \left[-\frac{\partial u}{\partial p_i}''(x;p) + r(p) \frac{\partial u}{\partial p_i}(x;p) \right] \mathrm{d}x,
\end{aligned}$$

where the last line follows by integrating by parts twice (boundary terms vanish because $\frac{\partial u}{\partial p_i}$ and z are zero at a and b)

(The adjoint equation is defined based on this "integration by parts" relationship to the primal equation)

Adjoint-Based Method

Also, recalling the derivative of the primal problem with respect to p_i:

$$-\frac{\partial u}{\partial p_i}''(x;p) + r(p)\frac{\partial u}{\partial p_i}(x;p) = -\frac{\partial r}{\partial p_i}u(x;p),$$

we get

$$\frac{\partial \mathcal{G}(p)}{\partial p_i} = -\frac{\partial r}{\partial p_i}\int_a^b z(x;p)u(x;p)\mathrm{d}x$$

Therefore, we only need to solve two differential equations (primal and adjoint) to obtain $\nabla \mathcal{G}(p)$!

For more complicated PDEs the adjoint formulation is more complicated but the basic ideas stay the same