Applied Mathematics 205

Unit IV: Nonlinear Equations and Optimization

Lecturer: Dr. David Knezevic

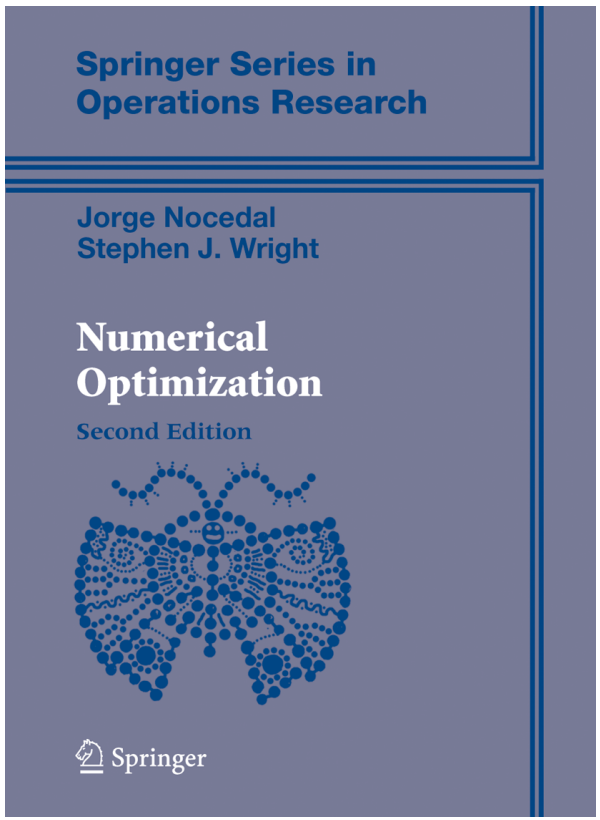
Unit IV: Nonlinear Equations and Optimization

Chapter IV.4: Survey of Optimization Methods

Numerical Optimzation

Numerical Optimization is a very large and important field; we do not have time to go into a great deal of depth

For more details, there are many good references on this area, for example: Numerical Optimization, by Nocedal and Wright



Unconstrained Optimization

Steepest Descent

We first consider the simpler case of unconstrained optimization (as opposed to constrained optimization)

Perhaps the simplest method for unconstrained optimization is steepest descent

Key idea: The negative gradient $-\nabla f(x)$ points in the "steepest downhill" direction for f at x

Hence an iterative method for minimizing f is obtained by following $-\nabla f(x_k)$ at each step

Question: How far should we go in the direction of $-\nabla f(x_k)$?

Steepest Descent

We can try to find the best step size via a subsidiary (and easier!) optimization problem

For a direction $s \in \mathbb{R}^n$, let $\phi : \mathbb{R} \to \mathbb{R}$ be given by

$$\phi(\eta) = f(x + \eta s)$$

Then minimizing f along s corresponds to minimizing the one-dimensional function ϕ

This process of minimizing f along a line is called a line search[1]

[1] The line search can itself be performed via Newton's method, as described for $f : \mathbb{R}^n \to \mathbb{R}$ shortly, or via a Matlab built-in function

Steepest Descent

Putting these pieces together leads to the steepest descent method:

1: choose initial guess x_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: $\quad s_k = -\nabla f(x_k)$
4: $\quad$ choose η_k to minimize $f(x_k + \eta_k s_k)$
5: $\quad x_{k+1} = x_k + \eta_k s_k$
6: **end for**

However, steepest descent often converges very slowly

Convergence rate is linear, and scaling factor can be arbitrarily close to 1

(You will implement steepest descent in Assignment 5)

Newton's Method

We can get faster convergence by using more information about f

Note that $\nabla f(x^*) = 0$ is a system of nonlinear equations, hence we can solve it with quadratic convergence via Newton's method[2]

The Jacobian matrix of $\nabla f(x)$ is $H_f(x)$ and hence Newton's method for unconstrained optimization is:

1: choose initial guess x_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: solve $H_f(x_k)s_k = -\nabla f(x_k)$
4: $x_{k+1} = x_k + s_k$
5: **end for**

[2]Note that in its simplest form this algorithm searches for stationary points, not necessarily minima

Newton's Method

We can also interpret Newton's method as seeking stationary point based on a sequence of local quadratic approximations

For small δ we can use Taylor's theorem to obtain the following approximation of f near x_k:

$$f(x_k + \delta) \approx f(x_k) + \nabla f(x_k)^T \delta + \frac{1}{2}\delta^T H_f(x_k)\delta \equiv q(\delta)$$

We find a stationary point of q in the usual way:[3]

$$\nabla q(\delta) = \nabla f(x_k) + H_f(x_k)\delta = 0$$

This leads to $H_f(x_k)\delta = -\nabla f(x_k)$, as in the previous slide

[3] Recall I.4 for differentiation of $\delta^T H_f(x_k)\delta$

Newton's Method

Matlab example: Newton's method for minimization of Himmelblau's function

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

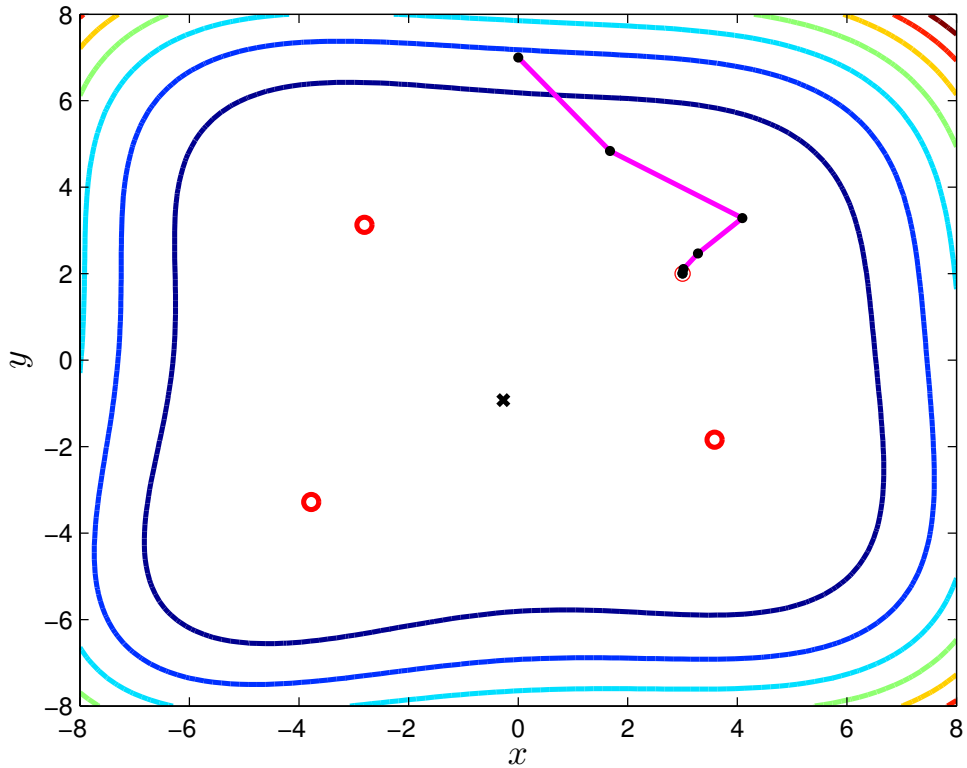
Local maximum of 181.617 at $(-0.270845, -0.923039)$

Four local minima (function value at each minimum is 0) at

$$(3, 2),\ (-2.805, 3.131),\ (-3.779, -3.283),\ (3.584, -1.841)$$

Newton's Method

Matlab example: Newton's method for minimization of Himmelblau's function



Newton's Method: Robustness

Newton's method generally converges much faster than steepest descent

However, Newton's method can be unreliable far away from a solution

To improve robustness during early iterations it is common to perform a line search in the Newton-step-direction

Also line search can ensure we don't approach a local max. as can happen with raw Newton method

The line search modifies the Newton step size, hence often referred to as a damped Newton method

Newton's Method: Robustness

Another way to improve robustness is with trust region methods

At each iteration k, a "trust radius" R_k is computed

This determines a region surrounding x_k on which we "trust" our quadratic approx.

We require $\|x_{k+1} - x_k\| \leq R_k$, hence constrained optimization problem (with quadratic objective function) at each step

Newton's Method: Robustness

Size of R_{k+1} is based on comparing actual change, $f(x_{k+1}) - f(x_k)$, to change predicted by the quadratic model

If quadratic model is accurate, we expand the trust radius, otherwise we contract it

When close to a minimum, R_k should be large enough to allow full Newton steps $\implies$ eventual quadratic convergence

Quasi-Newton Methods

Newton's method is effective for optimization, but it can be unreliable, expensive, and complicated

- Unreliable: Only converges when sufficiently close to a minimum
- Expensive: The Hessian H_f is dense in general, hence very expensive if n is large
- Complicated: Can be impractical or laborious to derive the Hessian

Hence there has been much interest in so-called quasi-Newton methods, which do not require the Hessian

Quasi-Newton Methods

General form of quasi-Newton methods:

$$x_{k+1} = x_k - \alpha_k B_k^{-1} \nabla f(x_k)$$

where α_k is a line search parameter and B_k is some approximation to the Hessian

Quasi-Newton methods generally lose quadratic convergence of Newton's method, but often superlinear convergence is achieved

We now consider some specific quasi-Newton methods

BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method is one of the most popular quasi-Newton methods:

1: choose initial guess x_0
2: choose B_0, initial Hessian guess, e.g. $B_0 = \mathrm{I}$
3: **for** $k = 0, 1, 2, \ldots$ **do**
4: solve $B_k s_k = -\nabla f(x_k)$
5: $x_{k+1} = x_k + s_k$
6: $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$
7: $B_{k+1} = B_k + \Delta B_k$
8: **end for**

where

$$\Delta B_k \equiv \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$$

BFGS

We won't go into details of the rationale behind the B_k updating scheme...

Basic idea is that B_k accumulates second derivative information on successive iterations, eventually approximates H_f well

(You will implement BFGS in Assignment 5, converges faster than steepest descent, slower than Newton)

BFGS

BFGS (+ trust region) is implemented in Matlab's `fminunc` function, e.g.

```
x0 = [5;5];
options = optimset('GradObj','on');
[x,fval,exitflag,output] = ...
  fminunc(@himmelblau_function,x0,options);
```

where `himmelblau_function` is given by:

```
function [f,grad] = himmelblau_function(x)

f = (x(1,:).^2 + x(2,:) - 11).^2 + (x(1,:) + x(2,:).^2 - 7).^2;
grad = [4*x(1,:).*(x(1,:).^2+x(2,:)-11) + 2*(x(1,:)+x(2,:).^2-7)
        2*(x(1,:).^2+x(2,:)-11) + 4*x(2,:).*(x(1,:)+x(2,:).^2-7)];
```

`fminunc` with starting point $x_0 = [5, 5]^T$ finds the minimum $x^* = [3, 2]^T$

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: $s_0 = -g_0$
4: **for** $k = 0, 1, 2, \ldots$ **do**
5: choose η_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**

Conjugate Gradient Method

Again, detailed derivation of CG is outside scope of AM205...

But we can see similarities to gradient descent, e.g. in lines 5 and 6

Difference is that in CG, the search direction s_{k+1} is modified in line 9 based on the previous search direction

This means CG has "memory" of past search directions, and hence tends to perform much better than gradient descent

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 IV.3 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

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

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

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

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 $i = n+1, \ldots, n+m$, we have

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

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

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

... and the final $m \times m$ "bottom right" block is just zero (differentiation of $g_i(x)$ wrt 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[4])

[4] 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 $k = 0, 1, 2, \ldots$

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

Next we demonstrate that this can again be re-interpreted as a sequence of quadratic minimization problems

Sequential Quadratic Programming

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 k^{th} Newton step

Hence at each step of Newton's method, we solve a minimization problem with quadratic objective function and linear constraints

An optimization problem of this type is called a quadratic program

This is why applying Newton's method to $\mathcal{L}(x, \lambda) = 0$ is called Sequential Quadratic Programming (SQP)

Sequential Quadratic Programming

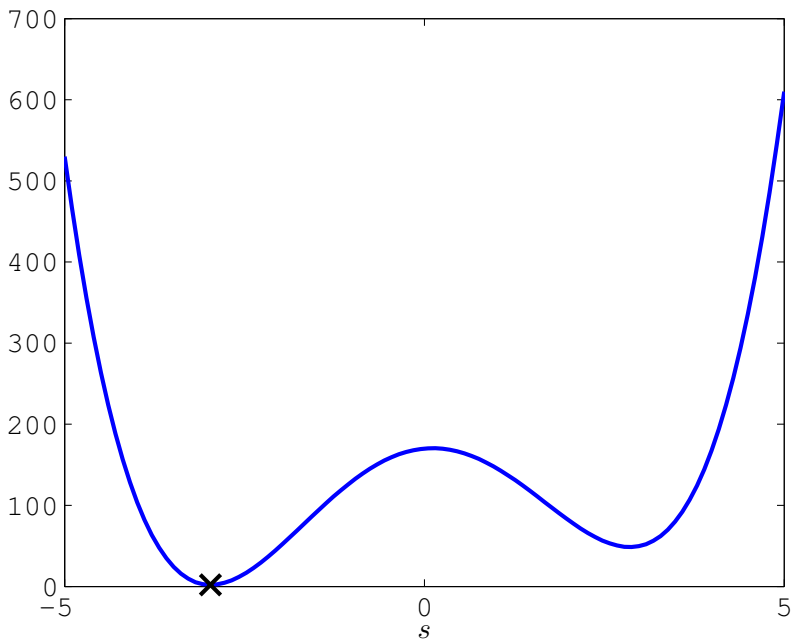
SQP is an important method, and there are many issues to be considered to obtain an efficient and reliable implementation:

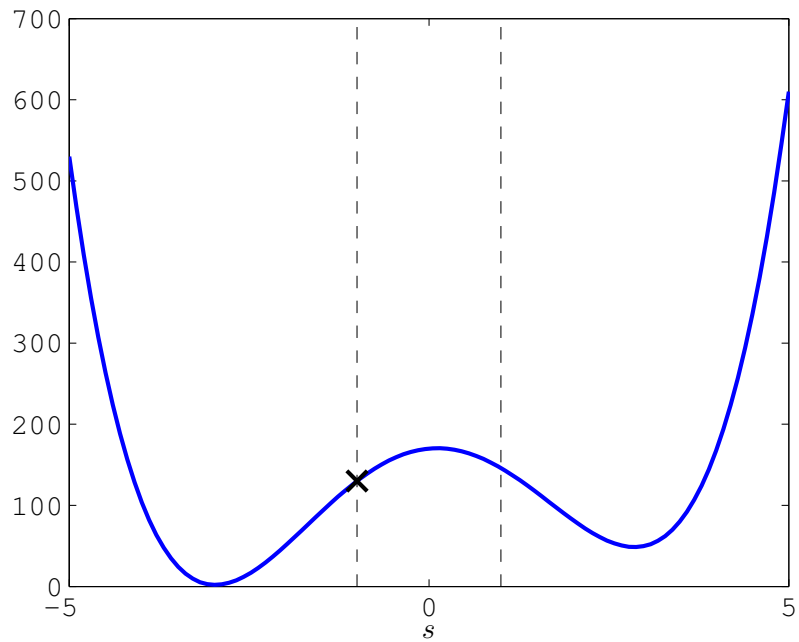
- Efficient solution of the linear systems at each Newton iteration — matrix block structure can be exploited
- Quasi-Newton approximations to the Hessian (as in the unconstrained case)
- Trust region, line search etc to improve robustness
- Treatment of constraints (equality and inequality) during the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

SQP is implemented in Matlab's `fmincon` function, which handles equality and inequality constraints

Sequential Quadratic Programming

Matlab example: `fmincon` for equality and inequality constrained optimization of Himmelblau's function





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

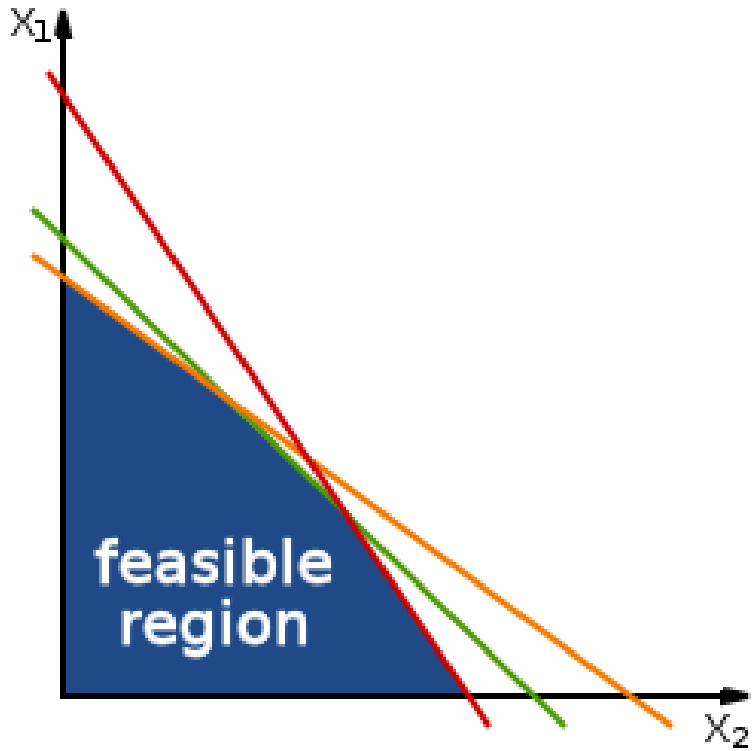
The feasible region is a convex polyhedron[5]

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

[5] 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[6]

Simplex method is generally very efficient, typically only requires a small subset of the vertices to be checked

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

[6] Developed by Dantzig, published in 1947

Linear Programming

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

But worst-case behavior is very rare; in practice computational work typically grows linearly with the number of variables

A different approach, called interior point method, was developed in the 1980s and in the worst case cost growth is polynomial

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

Linear Programming

Matlab example: Using `linprog`, solve the linear program[7]:

$$\min_x f(x) = -5x_1 - 4x_2 - 6x_3$$

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!)

[7] From Mathworks website

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 "wind" 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,[8] 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:

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

[8] 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

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

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:

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

In the Assignment, you are asked to use perform PDE-based optimization using `fmincon`

In particular, you were instructed not to pass gradient or Hessian data to `fmincon`

But of course `fmincon` needs some derivative information, so it uses 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 can see this effect from `output` of `fmincon`/`fminunc`, e.g. minimizing Himmelblau[9] with `fminunc` and F.D. gradient:

iterations:	9
function calls:	32

[9] For Himmelblau's function we have $n = 2$

PDE Constrained Optimization

Whereas minimizing Himmelblau with `fminunc` and `optimset('GradObj','on')` gives

iterations:	7
function calls:	8

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

(These are advanced topics that we will only touch on briefly...)

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:[10]

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

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

[10] Note that the "point-value" output in the Assignment is not an integral-based output (unless we allow σ to be a Dirac delta function)

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...