Optimization

This lecture explores the optimization tools available in Matlab. We begin with a light discussion on methods of minimization.

1 The Optimization Problem

Consider an optimization problem where we need to find the minimum value of a function $f : \mathbb{R} \mapsto \mathbb{R}$ on some subset of its domain:

$$\min_{x \in D} f(x)$$

We are interested in solving this problem. However, it is not clear that a solution to this problem exists. We know that if f is a continuous function and D is a compact set, then by the Weierstrass theorem $\exists x^* \in D : f(x^*) \leq f(x), \forall x \in D$. Under these conditions, we know that a solution to the problem exists, but we still need a way to find it. If the function f is not continuous, or D is not a compact set, then a solution may or may not exist.

Even though we cannot always guarantee that a solution exists, if a solution does exist, then it must satisfy a necessary condition. If x_0 is a point where f is minimized (or maximized) and f is differentiable at x_0, then $f'(x_0) = 0$ (see Fermat's theorem on stationary points). This necessary condition motivates many of the optimization algorithms.

The algorithms for finding the argument that minimizes a function are divided in two types: direct search methods and gradient-based methods. Direct search methods do not rely on the derivative of a function, and so can be applied to non differentiable functions. These methods directly search for optimal points across the domain of a function. An example of a direct search method is the Golden-section search. Gradient-based methods use the derivative of a function to find a value x such that $f'(x) = 0$. In these algorithms, the gradient can be explicitly computed or approximated numerically. Examples of gradient-based algorithms are: Bisection method, Newton-Raphson method and Secant method.

These optimization methods apply to functions of a single variable. However, they are building blocks for optimization methods for functions of many variables. When it comes to optimization algorithms for functions of many variables, we can still divide the algorithms in the same two types. Examples of direct search methods for multivariate functions are: Powell's method and Nelder-Mead method. Some of the optimization methods that use the gradient are: Newton's method, DFP method, BFGS method and the Gradient Descent method.

The methods above apply to unconstrained optimization problems. However, in practice we usually have to deal with optimization problems that are subject to constraints:

$$\begin{aligned} \underset{x\in\mathbb{R}^d}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & f_i(x) \leq b_i, i = 1, \ldots, d \end{aligned}$$

For constrained problems, some of the optimization methods are: Penalty function method, Augmented Lagrangian method and Sequential quadratic programming. These methods either transform the constrained problem into an unconstrained problem with an added term that penalizes values that are outside the original constraints, or take extra care when looking through the domain to make sure the points satisfy the constrains.

Fortunately, we do not need to implement these algorithms, as most of them are already implemented in Matlab (and in various other programming languages). A reference for all of the optimization algorithms mentioned above is Arora (2015), which also provides the code for these algorithms. On the next sections we will discuss how to use the optimization methods available in Matlab, available via the Optimization Toolbox (should already be installed if you also installed the Econometrics Toolbox).

2 Unconstrained Optimization (Gradient Based)

The function `fminunc` solves the optimization problem:

$$\min_x f(x)$$

where $f : \mathbb{R}^d \mapsto \mathbb{R}$ is a function of many variables that returns a scalar. Remember that maximizing a function g is equivalent to minimizing $-g$, so this solver can deal with minimization and maximization.

To use `fminunc` we need to supply it a function handle and an initial point. It works with functions of a single variable:

```
% polynomial example
f = @(x) x.^2;
x0 = 100;
[x, fx] = fminunc(f, x0);
% x: the argmin of the function
% fx: the value of the function at the minimizer
% the value of x is numerically close to zero (same for f'(x)
   )
```

And also functions of many variables:

```
% polynomial example on R2
f = @(x) 3*x(1).^2 + 2*x(1)*x(2) + x(2).^2 - 4*x(1) + 5*x(2);
% surface plot over a large region
x = linspace(-10000, 10000)';
y = linspace(-10000, 10000)';
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
```

```
for i = 1:length(Z)
    z(i) = f(Z(i, :));
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
hold on;
% find minimum
x0 = [1, 1];
[x, fx] = fminunc(f, x0);
% draw minimum on plot
scatter3(x(1), x(2), fx, 60, 'ro', 'filled');
```

We used `scatter3` to add a point at the minimum. The surface plot is shown in Figure 1.

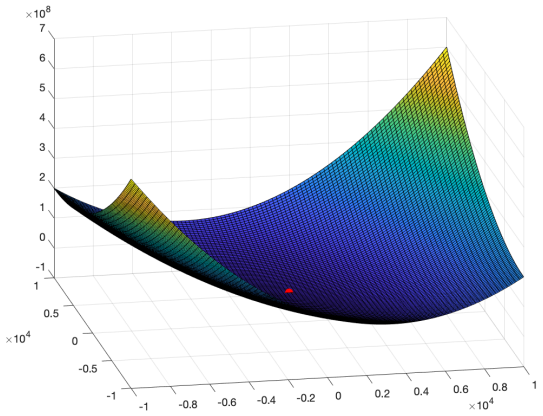

Figure 1: Surface of a Polynomial on $\mathbb{R}^2$ and its Minimum.

Notice that in the examples above we did not supply the gradient of the function to `fminunc`. In this case, Matlab is limited to numerically approximating the derivative, which can be unstable and is slow. When no gradient is supplied, `fminunc` uses the Quasi-Newton method and approximates the derivative with finite differences. However, we can improve `fminunc` by also providing the gradient of the function. In this case, `fminunc` can also use the Trust-Region method.

To supply the gradient to `fminunc`, the function we want to minimize should output two values: the first is the objective function value (scalar), and the second is the gradient (vector) of the function.

```
% rosenbrock.m
function [f, gradient] = rosenbrock(x)
% rosenbrock computes the Rosenbrock function
% reference: https://en.wikipedia.org/wiki/
   Rosenbrock_function
f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
gradient = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
            200*(x(2)-x(1)^2)];

```

```
% check if user expects more than 1 output (i.e., also wants
   the gradient)
if nargout > 1
    gradient = [-400*(x(2)-x(1)^2)*x(1)-2*(1-x(1));
                200*(x(2)-x(1)^2)];
end
```

We use `nargout` to check how many outputs are expected.

```
% in this case nargout is 1
f = rosenbrock([1; 0])
% in this case nargout is 2
[f, gradient] = rosenbrock([1; 0])
```

Let's visualize this function:

```
x = linspace(-2, 2);
y = linspace(-3, 3);
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
for i = 1:length(Z)
    z(i) = rosenbrock(Z(i, :));
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
hold on;
% actual minimum
scatter3(1, 1, 0, 60, 'ro', 'filled');
```

The graph of the function is shown in Figure 2.

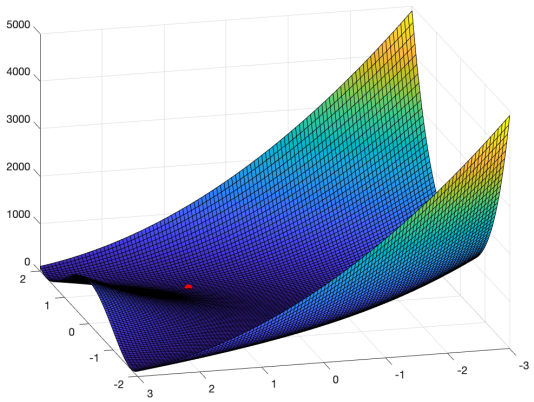

Figure 2: Rosenbrock Function and Global Minimum.

We can now call `fminunc` and pass it an `options` argument, which specifies that the gradient of the function is also available. The `options` argument is created with the function `optimoptions`.

```
% set up options for fminunc
% The first argument is the optimizer we are using, in this case,
% 'fminunc'. The other arguments are name-value pairs.
options = optimoptions('fminunc', ...
                       'SpecifyObjectiveGradient', true, ...
                       'Algorithm', 'trust-region');
[x, fx] = fminunc(@rosenbrock, [-1, 2], options);
```

We can also use the `options` argument to display the algorithm iterations as they occur:

```
options = optimoptions('fminunc', ...
                       'SpecifyObjectiveGradient', true, ...
                       'Algorithm', 'trust-region', ...
                       'Display', 'iter');
[x, fx, exitflag, output] = fminunc(@rosenbrock, [-1, 2], options);
```

The `output` is a `struct` containing information about the optimization, and the `exitflag` indicates a number representing the type of solution obtained (1 for local minima, see `exitflag` for the meaning of the other values).

During the optimization process, `fminunc` computes the gradient and the Hessian of the function. We can obtain the gradient and the Hessian at the end of the minimization process:

```
[x, fx, exitflag, output, grad, hess] = fminunc(@rosenbrock, [-1, 2], options);
```

There are other options that we can set:

- `MaxIterations`: maximum number of iterations (default is 400);
- `OptimalityTolerance`: tolerance for $f'(x) = 0$ (default is 10^{-6})
- `StepTolerance`: tolerance for the step size (default is 10^{-6})
- `PlotFcn`: plot the function value as it is optimized
- `UseParallel`: approximates the gradient in parallel

For example, with `PlotFcn` we can see the evolution of the optimization graphically:

```
plots = ["optimplotx", "optimplotfval", ...
         "optimplotstepsize", "optimplotfirstorderopt"];
options = optimoptions('fminunc', ...
                       'SpecifyObjectiveGradient', true, ...
                       'Algorithm', 'trust-region', ...
                       'Display', 'iter', ...
                       'PlotFcn', plots);
[x, fx, exitflag, output] = fminunc(@rosenbrock, [-1, 2], options);
```

See Figure 3.

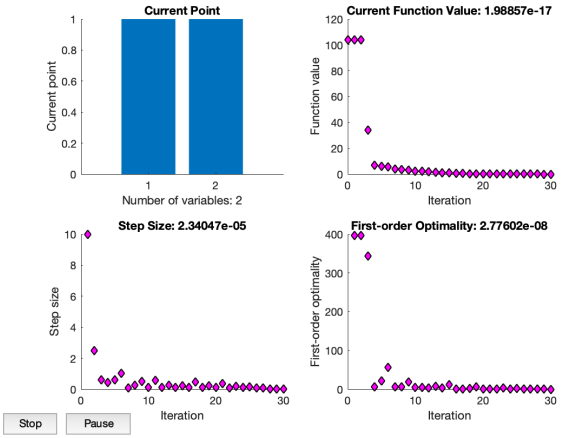

Figure 3: Function Value and Inputs During Minimization.

A complete list of options is available at the `fminunc` reference page.

3 Unconstrained Optimization (Direct Search)

The function `fminsearch` also solves an unconstrained minimization problem, but uses a direct search method that does not rely on the gradient of the function being minimized. Specifically, `fminsearch` uses the Nelder-Mead algorithm.

The usage of `fminsearch` is similar to `fminunc`:

```
[x, fx] = fminsearch(@rosenbrock, [-1, 2]);
```

To specify extra options, however, we use the function `optimset` (it is different from the one used with `fminunc`):

```
options = optimset('Display', 'iter');
[x, fx, exitflag, output] = fminsearch(@rosenbrock, [-1, 2],
   options);
```

Notice that `fminsearch` does not return the gradient and hessian of the function at the minimizer, since it does not use derivatives.

4 Constrained Optimization

The function `fmincon` solves the following constrained minimization problem:

$$\begin{aligned} \underset{x\in\mathbb{R}^d}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & \begin{cases} c(x) \leq 0 \\ ceq(x) = 0 \\ A \cdot x \leq b \\ Aeq \cdot x = beq \\ lb \leq x \leq ub \end{cases} \end{aligned}$$

Where: $f : \mathbb{R}^d \mapsto \mathbb{R}$, $c : \mathbb{R}^d \mapsto \mathbb{R}^{k_1}$, $ceq : \mathbb{R}^d \mapsto \mathbb{R}^{k_2}$, $A \in \mathbb{R}^{k_3 \times d}$ and $b \in \mathbb{R}^{k_3 \times 1}$, $Aeq \in \mathbb{R}^{k_4 \times d}$ and $beq \in \mathbb{R}^{k_4 \times 1}$, and x, lb and ub are vectors of length d.

We can call `fmincon` using several different configurations, which we explore in the next subsections.

4.1 Linear Inequality Constraint

Let's minimize Rosenbrock's function with the constraint that $x_1 + 2x_2 \leq 1$:

```
%% Let's minimize Rosenbrock's function with an additional
% constraint
A = [1 2];
x0 = [-1; 2];
b = 1;
%% Visualize the graph of the function and the linear
   constraint
x = linspace(-2, 2, 60);
y = linspace(-3, 3, 60);
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
for i = 1:length(Z)
    z(i) = rosenbrock(Z(i, :));
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
fig.FaceAlpha = 0.8;
colormap cool;
hold on;
% add constraint
X1 = [2;2;-2;-2];
X2 = [-3;-0.5;3/2;-3];
fill(X1, X2, 'green');
% actual minimum
scatter3(1, 1, 0, 60, 'ro', 'filled');
%% Minimize the function under the constraint
[x, fx] = fmincon(@rosenbrock, x0, A, b);
%% Add minimization result to figure
scatter3(x(1), x(2), fx, 60, 'ko', 'filled');
% add labels and legend
xlabel('x_1');
ylabel('x_2');
legend(["Rosenbrock's Function", "Linear Constraint", ...
        "Global Minimum", "Constrained Minimum"], ...
       "Location", "northwest");
```

We can visualize the result of the constrained minimization in Figure 4 below.

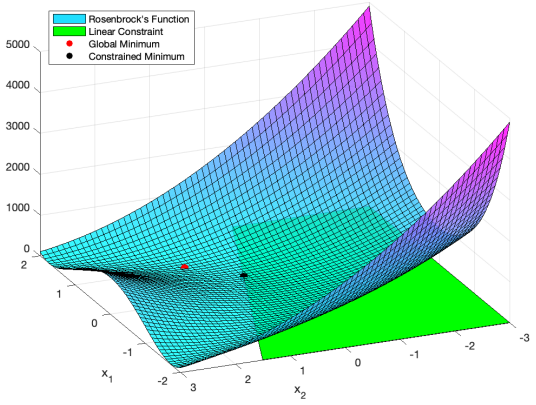

Figure 4: Minimization of the Rosenbrock's Function under a Linear Constraint.

4.2 Linear Inequality and Equality Constraints

Let's add a linear equality constraint:

```
%% Let's minimize Rosenbrock's function with two linear
   constraints
% inequality constraint: x(1) + 2*x(1) <= 1
A = [1 2];
b = 1;
% equality constraint: 2*x(1) + x(2) = 1
Aeq = [2 1];
beq = 1;
%% Visualize function and constraints
x = linspace(-2, 2, 60);                    % x1
y = linspace(-3, 3, 60);                    % x2
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
for i = 1:length(Z)
    z(i) = rosenbrock(Z(i, :));
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
fig.FaceAlpha = 0.8;
colormap cool;
hold on;
% add inequality constraint
X1 = [2;2;-2;-2];
X2 = [-3;-0.5;3/2;-3];
fill(X1, X2, 'green');
% add equality constraint
plot(x, 1- 2.*x, 'k', 'LineWidth', 2);
% actual minimum
scatter3(1, 1, 0, 60, 'ro', 'filled');
```

```
%% Minimize the function under the constraint
x0 = [0; 0.5];
[x, fx] = fmincon(@rosenbrock, x0, A, b, Aeq, beq);
%% Add minimization result to figure
scatter3(x(1), x(2), fx, 60, 'ko', 'filled');
% add labels and legend
xlim([-2, 2]);
ylim([-3, 3]);
xlabel('x_1');
ylabel('x_2');
legend(["Rosenbrock's Function", "Inequality Constraint", ...
        "Equality Constraint", "Global Minimum", ...
        "Constrained Minimum"], "Location", "northwest");
```

We can visualize the result of the constrained minimization in Figure 5 below.

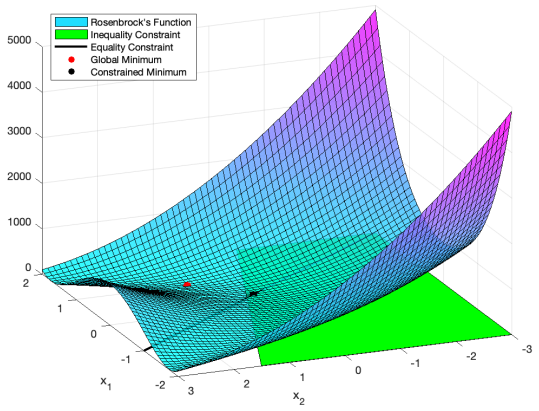

Figure 5: Minimization of the Rosenbrock's Function under a Linear Inequality Constraint and a Linear Equality Constraint.

4.3 Linear Equality Constraint

If we did not want to specify the linear inequality constraint, then we can pass an empty matrix `[]` as the inputs for `A` and `b`:

```
%% Let's minimize Rosenbrock's function with a linear
   equality constraint
% equality constraint: 2*x(1) + x(2) = 1
Aeq = [2 1];
beq = 1;
%% Visualize function and constraints
x = linspace(-2, 2, 60);                % x1
y = linspace(-3, 3, 60);                % x2
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
for i = 1:length(Z)
    z(i) = rosenbrock(Z(i, :));
```

```
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
fig.FaceAlpha = 0.8;
colormap cool;
hold on;
% add equality constraint
plot(x, 1- 2.*x, 'k', 'LineWidth', 2);
% actual minimum
scatter3(1, 1, 0, 60, 'ro', 'filled');
%% Minimize the function under the constraint
x0 = [0.5; 0];
[x, fx] = fmincon(@rosenbrock, x0, [], [], Aeq, beq);
%% Add minimization result to figure
scatter3(x(1), x(2), fx, 60, 'ko', 'filled');
% add labels and legend
xlim([-2, 2]);
ylim([-3, 3]);
xlabel('x_1');
ylabel('x_2');
legend(["Rosenbrock's Function", "Equality Constraint", "
   Global Minimum", ...
        "Constrained Minimum"], "Location", "northwest");
```

We can visualize the result of the constrained minimization in Figure 6 below.

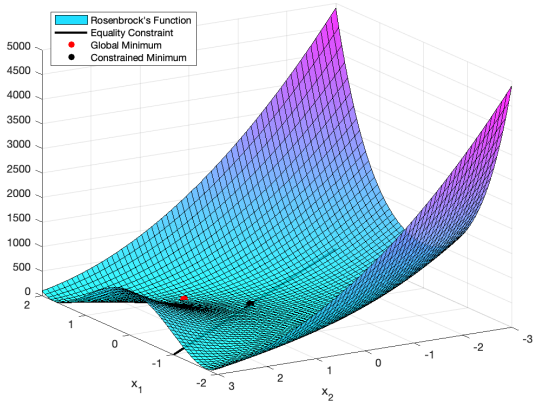

Figure 6: Minimization of the Rosenbrock's Function under a Linear Equality Constraint.

4.4 Bound Constraints

We now consider bound constraints: $lb \leq x \leq ub$.

```
%% Specify a function
f = @(x) (1+x(1)/(1+x(2)) - 3*x(1)*x(2) + x(2)*(1+x(1)));
%% Visualize f
x = linspace(-0.5, 1.5, 60);                % x1
y = linspace(-0.5, 3, 60);                 % x2
```

```
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
for i = 1:length(Z)
    z(i) = f(Z(i, :));
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
hold on;
xlabel('x1');
ylabel('x2');
%% Define bound constraint
% positive values of x
lb = [0; 0];
ub = [1; 2];
%% Minimize function
x0 = [0.5, 1];
[x, fx] = fmincon(f, x0, [], [], [], [], lb, ub);
%% Visualize minima
scatter3(x0(1), x0(2), f(x0), 60, 'ko', 'filled');
scatter3(x(1), x(2), fx, 60, 'ro', 'filled');
%% Minimize function
x0 = x0./5;
[x, fx] = fmincon(f, x0, [], [], [], [], lb, ub);
%% Visualize minima
scatter3(x0(1), x0(2), f(x0), 60, 'ko', 'filled');
scatter3(x(1), x(2), fx, 60, 'ro', 'filled');
legend(["Graph of f", "Initial Guess", "Local Minimum", ...
        "Another Initial Guess", "Another Local Minimum"], ...
       "Location", "northwest");
```

Figure 7 below displays the function and the local minima we found in the constrained region.

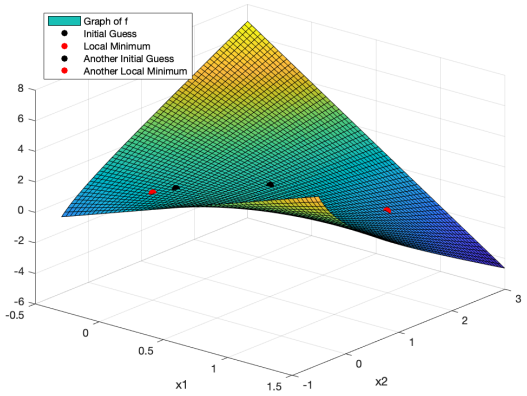

Figure 7: Minimization of a Function Under Bound Constraints.

We can find the global minima in this region by comparing the objective function value at the two local minima.

4.5 Nonlinear Constraint

Let's find the minimum of the Rosenbrock's function when the domain is constrained to a circle. First, we specify the nonlinear constraint by creating a new function. This function must take an input x, and must have two outputs: $c(x)$ and $ceq(x)$. If the nonlinear inequality constraint or the nonlinear equality constraint are not used, then the function should simply output `[]`.

Remember that a point (x, y) is in a circle centered at (a, b) with radius r if:

$$\|(x, y) - (a, b)\| \leq r$$

Now, `fmincon` considers the nonlinear constraint to be of the form $c(x) \leq 0$. So we need to convert the inequality above to be of that form. We can also write the inequality in terms of squares instead of square-root, since computing the square of a number is faster than computing its square-root:

$$\begin{aligned} \|(x, y) - (a, b)\| \leq r &\iff (x-a)^2 + (y-b)^2 \leq r^2 \\ &\iff (x-a)^2 + (y-b)^2 - r^2 \leq 0 \end{aligned}$$

We can now write this inequality in a function:

```
% in_circle.m
function [c, ceq] = in_circle(point, center, radius)
% in_cirlce returns a negative number if the point is in the circle,
% otherwise returns a positive number
c = sum((point - center).^2) - radius^2; % <= 0
ceq = [];                                % == 0
```

Now, we can minimize Rosenbrock's function subject to this nonlinear constraint:

```
%% Minimize function
fun = @(x) (in_circle(x, [0.5 0.5], 0.5));
x0 = [0.5 1];
[x_min, f_min] = fmincon(fun, x0, [], [], [], [], [], [], fun);
%% Visualize problem
x = linspace(-0.5, 1.5, 60);                % x1
y = linspace(-0.5, 1.5, 60);                % x2
[X, Y] = meshgrid(x, y);
Z = [X(:) Y(:)];
z = zeros(length(Z), 1);
for i = 1:length(Z)
    z(i) = rosenbrock(Z(i, :));
end
z = reshape(z, length(y), length(x));
fig = surf(x, y, z);
fig.FaceAlpha = 0.6;
```

```
colormap cool;
hold on;
r = rectangle('Position', [0 0 1 1], 'Curvature', [1 1], ...
              'FaceColor', 'white');
scatter3(x0(1), x0(2), rosenbrock(x0), 120, 'ko', 'filled');
scatter3(x_min(1), x_min(2), f_min, 120, 'ro', 'filled');
xlabel('x1');
ylabel('x2');
xlim([-0.5 1.5]);
ylim([-0.5 1.5]);
legend(["Rosenbrock's Function", "Initial Guess", "Local
   Minimum"], ...
       'location', 'northwest');
```

We used the function `rectangle` to draw the circle. The optimization result is shown in Figure 8.

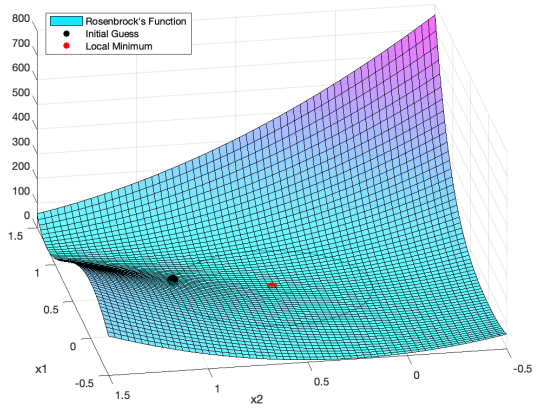

Figure 8: Minimization of Rosenbrock's Function Under a Nonlinear Constraint.

4.6 Options and Algorithms

There are five algorithm options for `fmincon`: `interior-point` (default), `trust-region-reflective`, `sqp`, `sqp-legacy` and `active-set`. The links above lead to the descriptions of each algorithm. The general reasoning for each of the algorithms is the following:

- Interior Point: Large-scale algorithm (uses sparse matrices, less memory). Constraints are satisfied at all times.
- Trust Region Reflective: Large-scale algorithm (uses sparse matrices, less memory). Requires the gradient of the function. Can only have bound constraints or linear equality constraints, but not both.
- SQP: Medium-scale algorithm (uses dense matrices, uses more memory, more accurate). Constraints are satisfied at all times. The legacy version is slower and uses more memory.
- Active Set: Medium-scale algorithm (uses dense matrices, uses more memory, more accurate). Is able to take large steps. Can deal with non-smooth constraints.

We can choose the algorithm by setting options for `fmincon` using `optimoptions`.

```
% Minimize Rosenbrock's on a unit circle
unit_circle = @(x) (in_circle(x, [0; 0], 1));
x0 = [0,0];
% Set options to display minimization progression
options = optimoptions('fmincon', 'Display', 'iter');

% Minimize with the default algorithm
options.Algorithm = 'interior-point';
[x_min, f_min] = fmincon(@rosenbrock,x0,[],[],[],[],[],[],
   unit_circle,options)

% Minimize with the sqp algorithm
options.Algorithm = 'sqp';
[x_min, f_min] = fmincon(@rosenbrock,x0,[],[],[],[],[],[],
   unit_circle,options)

% Minimize with the active-set algorithm
options.Algorithm = 'active-set';
[x_min, f_min] = fmincon(@rosenbrock,x0,[],[],[],[],[],[],
   unit_circle,options)

% Minimize with the trust-region-reflective algorithm
% will not work due to constraints (and gradient)
options.Algorithm = 'trust-region-reflective';
[x_min, f_min] = fmincon(@rosenbrock,x0,[],[],[],[],[],[],
   unit_circle,options)
```

4.7 Including the Gradient

To include the gradient we specify the objective function to return two outputs. The first is the value of the function, and the second is the gradient of the function. We can then specify the option `SpecifyObjectiveGradient` to be `true`:

```
% Minimize Rosenbrock's function over positive values
lb = [0 0];
ub = [Inf Inf];
x0 = [3 3];
% Set options to display minimization progression
options = optimoptions('fmincon', 'Display', 'iter', ...
                       'SpecifyObjectiveGradient', true);

% Minimize with the default algorithm
options.Algorithm = 'interior-point';
[x_min, f_min] = fmincon(@rosenbrock,x0,[],[],[],[],lb,ub,[],
   options)

```

```
% Minimize with the trust-region-reflective algorithm (requires the
% gradient)
options.Algorithm = 'trust-region-reflective';
[x_min, f_min] = fmincon(@rosenbrock,x0,[],[],[],[],lb,ub,[],options)
```

5 Solving Nonlinear Equations

The function `fsolve` can be used to solve a system of nonlinear equations:

$$\begin{cases} f_1(x) &= 0 \\ f_2(x) &= 0 \\ & \vdots \\ f_n(x) &= 0 \end{cases}$$

where x is a vector and $n \geq 1$. The function `fsolve` starts with an initial guess x_0 and attempts to find x such that all functions evaluate to zero.

Let's solve a system of equations with `fsolve`. Consider the equations:

$$\begin{cases} \mathrm{e}^{-\mathrm{e}^{x_1+x_2}} &= x_2(1+x_1^2) \\ x_1\cos x_2 + x_2 \sin x_1 &= \frac{1}{2} \end{cases} \iff \begin{cases} \mathrm{e}^{-\mathrm{e}^{x_1+x_2}} - x_2(1+x_1^2) &= 0 \\ x_1\cos x_2 + x_2 \sin x_1 - \frac{1}{2} &= 0 \end{cases}$$

We can write the system above as:

$$\begin{cases} f_1(x) = 0 \\ f_2(x) = 0 \end{cases} \quad \text{where} \quad \begin{cases} f_1(x) &= \mathrm{e}^{-\mathrm{e}^{x_1+x_2}} - x_2(1+x_1^2) \\ f_2(x) &= x_1\cos x_2 + x_2 \sin x_1 - \frac{1}{2} \end{cases}$$

To use `fsolve` we need to write the system of equations in a function. The function should output a vector, representing the output of each equation in the system:

```
% system2d.m
function f = system2d(x)
f1 = exp(-exp(sum(x))) - x(2)*(1+x(1)^2);
f2 = x(1)*cos(x(2)) + x(2)*sin(x(1)) - 0.5;
f = [f1; f2];
```

We can now call `fsolve` to attempt to solve the system:

```
% Solve system
x0 = [5; -5];
[x_root, fval_root] = fsolve(@system2d, x0);
% Visualize system and solution
f1 = @(x, y) (exp(-exp(x + y)) -y*(1+x^2));
f2 = @(x, y) (x*cos(y) + y*sin(x) - 0.5);
fsurf(f1, 'FaceColor', 'red', 'FaceAlpha', 0.6);
hold on;
fsurf(f2, 'FaceColor', 'green', 'FaceAlpha', 0.6);
```

```
scatter3(x_root(1), x_root(2), 0, 400, 'ko', 'filled');
xlabel('x1');
ylabel('x2');
```

We can visualize the solution of the system in Figure 9. Notice the use of `fsurf` to plot the functions.

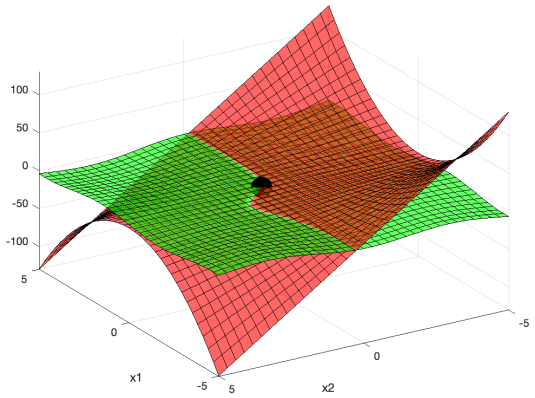

Figure 9: Solution of System of Nonlinear Equations.

We can display the progression of `fsolve` by specifying the `Display` option with the function `optimoptions`:

```
options = optimoptions('fsolve', 'Display', 'iter');
x0 = [5; -5];
[x_root, fval_root] = fsolve(@system2d, x0, options);
```

It is possible to use `fsolve` with functions where the input is a matrix. For example, to find the square-root of a matrix A, we could create the following system:

$$f(X) = X * X - A$$

We need to write the function above in a file:

```
% sqrtA.m
function X = sqrtA(X0, A)
X = X0*X0 - A;
% function must return a vector
X = X(:);
```

We can now use `fsolve` to find the square-root of a matrix A:

```
% find the sqrt of A
A = eye(2);
X0 = rand(2);
X = fsolve(@(X) (sqrtA(X, A)), X0)
```

Observe that if a system has multiple solutions, `fsolve` will only find one.

6 Finding the Root of a Nonlinear Function

The function `fzero` can be used to find the root of a single nonlinear function. That is, find the value x such that the nonlinear function f satisfies $f(x) = 0$. Like `fsolve`, `fzero` will only find one root. The function `fzero` uses a method similar to the bisection method, which requires the function f to have a change in sign. If f does not change sign, then `fsolve` will not be able to find its root. For example, `fsolve` cannot find the root of $f(x) = x^2$.

The initial guess taken by `fzero` can be a scalar or a vector with two values. If it is a scalar, say x_0, then `fzero` tries to find another point x_1 with the opposite sign of $f(x_0)$, and then applies the bisection method to shrink the interval until a solution is reached. If it is a vector, say $\begin{bmatrix} x_0 & x_1 \end{bmatrix}$, then `fzero` checks that $f(x_0)$ and $f(x_1)$ have different signs, and then applies the bisection method. However, it shows an error if the signs are not different.

```
% visualize polynomial
fun = @(x) x.^3 - 2*x.^2 + 3*x - 10;
fplot(fun)
% find root from initial guess (scalar)
x_min = fsolve(fun, -5);
hold on;
scatter(x_min, fun(x_min), 'ko', 'filled');
% find root from initial guess (vector)
x_min = fsolve(fun, [-5, 5]);
hold on;
scatter(x_min, fun(x_min), 'ro', 'filled');
```

7 Assignment

Problem 1 *(Conditional Maximum Likelihood Estimator) Consider the model:*

$$Y = \beta_0 + \beta_1 X + \varepsilon$$

where $\varepsilon \overset{d}{\sim} \mathcal{N}(0, \sigma^2)$*. We can write the conditional density of* $Y|X$ *as the density of a normal (due to* ε*) with its mean shifted by* $\beta_0 + \beta_1 X$*:*

$$f_{\{Y|X\}}(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-(\beta_0+\beta_1 X))^2}{2\sigma^2}}$$

We can use this density to write the (conditional) likelihood given i.i.d. observations $\{Z_i \equiv (Y_i, X_i)\}_{i=1}^n$*:*

$$L(\beta_0, \beta_1, \sigma; Z) = \Pi_{i=1}^n f_{\{Y|X\}}(y; \beta_0, \beta_1, \sigma)$$

Thus, we can write the (conditional) maximum likelihood estimator for β_0*,* β_1 *and* σ *as:*

$$(\hat{\beta}_0, \hat{\beta}_1, \hat{\sigma}) = \underset{\beta_0, \beta_1, \sigma}{\operatorname{argmax}} \sum_{i=1}^n -\frac{1}{2} ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}(y_i - \beta_0 - \beta_1 x_i)^2$$

Given appropriate values for β_0 *and* β_1*, simulate* 1000 *observations for* X *from a normal distribution. Graph the likelihood function (or log-likelihood) given the observations.*

Problem 2 *(continuation of previous problem) Using the simulated data from the previous exercise, estimate the parameters with the* `linreg_ols` *function. Now, estimate the parameters with MLE. You can use* `fminunc` *or* `fminsearch` *to find the estimates. Compare the results.*

Problem 3 *(continuation of previous problem) Compute the gradient of the log-likelihood function. Use* `fminunc` *and the gradient of the log-likelihood function to estimate the parameters via MLE. You can supply the option* `CheckGradients` *on a test-run to let Matlab check if the gradient you supplied is similar to the gradient computed numerically (good for checking if your code is correct).*

Problem 4 *(continuation of previous problem) We can also supply the Hessian to the* `fminunc` *function. To do so, we add a third output (the Hessian matrix) to the file of the function we are minimizing. Then, we set the option* `HessianFcn` *to* `'objective'`*. Use* `fminunc` *with the gradient and the Hessian of the log-likelihood function to estimate the parameters via MLE.*

Problem 5 *(Probit) The Probit model (see Example 7.3 of Hayashi (2000) is used to analyze data where the dependent variable is binary ($Y \in \{0,1\}$). In this context, we can write the conditional probability of Y as:*

$$\begin{cases} \mathbb{P}(Y=1|X;\theta) &= \Phi(X'\theta) \\ \mathbb{P}(Y=0|X;\theta) &= 1-\Phi(X'\theta) \end{cases}$$

where Φ is the cdf of the standard normal distribution, X is a vector of independent random variables and θ is a vector of unobserved parameters. It is possible to write the conditional probability of Y in a single equation, since Y can only take binary values:

$$\mathbb{P}(Y|X;\theta) = \Phi(X'\theta)^Y (1-\Phi(X'\theta))^{1-Y}$$

Given i.i.d. observations $\{(y_i, x_i)\}_{i=1}^n$, we can write the (conditional) log-likelihood function for $Z \equiv (Y, X)$ as:

$$l(\theta; Z) = \sum_{i=1}^{n} y_i ln\left(\Phi(x_i'\theta)\right) + (1-y_i) ln\left(1-\Phi(x_i'\theta)\right)$$

Given appropriate values for θ, simulate 1000 observations for X from a normal distribution. Graph the likelihood function (or log-likelihood) given the observations.

Problem 6 *(continuation of previous problem) Estimate the parameters using MLE and* `fminunc`*.*

Problem 7 *(Logit) The Logit model (see page 508 of Hayashi (2000)) is an alternative to the Probit model when it comes to analyzing binary data. It can be argued that the interpretation of the parameters is more direct in the case of the Logit model. We can write the conditional probability of Y as:*

$$\begin{cases} \mathbb{P}(Y=1|X;\theta) &= \Lambda(X'\theta) \\ \mathbb{P}(Y=0|X;\theta) &= 1-\Lambda(X'\theta) \end{cases}$$

where $\Lambda(v) = \frac{e^v}{1+e^v}$ *is the cdf of the logistic distribution,*X *is a vector of independent random variables and* θ *is a vector of unobserved parameters. The above is equivalent to:*

$$\mathbb{P}(Y|X;\theta) = \Lambda(X'\theta)^Y(1-\Lambda(X'\theta))^{1-Y}$$

Given i.i.d. observations $\{(y_i, x_i)\}_{i=1}^n$*, we can write the (conditional) log-likelihood function for* $Z \equiv (Y, X)$ *as:*

$$l(\theta; Z) = \sum_{i=1}^{n} y_i ln\left(\Lambda(x_i'\theta)\right) + (1-y_i) ln\left(1-\Lambda(x_i'\theta)\right)$$

Given appropriate values for θ*, simulate* 1000 *observations for* X *from a normal distribution. Graph the likelihood function (or log-likelihood) given the observations.*

Problem 8 *(continuation of previous problem) Estimate the parameters using MLE and* `fminunc`*.*

Problem 9 *(continuation of previous problem) (Optional) Derive the gradient and hessian of the log-likelihood in this case. (Hint: Equation (8.1.6) in Hayashi (2000)) Estimate the parameters using MLE and* `fminunc`*, while supplying the gradient and hessian.*

Problem 10 *(Dependent Observations) Consider a Gaussian AR(1) process (see page 546 of Hayashi (2000)):*

$$y_t = \alpha + \beta y_{t-1} + \varepsilon_t$$

where the $\varepsilon_t \overset{d}{\sim} \mathcal{N}(0, \sigma^2)$ *and are i.i.d, and* $|\beta| < 1$*. Simulate* 1000 *observations given an appropriate initial value for* y_0*.*

The log-likelihood in this case is given by:

$$l(\theta; Z) = \frac{1}{n}\sum_{t=1}^{n}\left[-\frac{1}{2}ln(2\pi) - \frac{1}{2}ln\left(\sigma^2\right) - \frac{1}{2\sigma^2}(y_t - \alpha - \beta y_{t-1})^2\right] + \\ + \frac{1}{n}\left[-\frac{1}{2}ln(2\pi) - \frac{1}{2}ln\left(\frac{\sigma^2}{1-\beta^2}\right) - \frac{(y_0 - \frac{\alpha}{1-\beta})^2}{2\frac{\sigma^2}{1-\beta^2}}\right]$$

Implement the log-likelihood and estimate the parameters via MLE.

Problem 11 *(Model Selection) We now consider the estimation of a linear regression via ordinary least-squares with the addition of a penalization term. The penalization term is a function of the magnitude of the parameters of the model. When we minimize the squared errors taking into account the penalization, some of the parameter estimates can be zero, leading to the estimation of a simpler model. For a good overview, read this page on Lasso.*

Consider the model:

$$Y = \beta_0 + \sum_{i=20} \beta_i X_i + \varepsilon$$

where $\varepsilon \overset{d}{\sim} \mathcal{N}(0, \sigma^2)$*, and each of the* X_i*'s are drawn from a normal distribution.*

Fix the values of the β_i*'s, but let* $\beta_1 = \beta_2 = 0$*. Simulate the data for* $\{(Y_i, X_{i,1}, \ldots, X_{i,20})\}_{i=1}^n$*, with* $n = 1000$*. Estimate the parameters via least-squares with* `linreg_ols`*.*

Problem 12 *(continuation of previous problem) Estimate the parameters by minimizing the squared errors with the penalization term added:*

$$\min_{\beta_0,\beta_1,\ldots,\beta_{20}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i,1} - \ldots - \beta_{20}x_{i,20})^2$$
$$\text{subject to} \sum_{i=0}^{20} |\beta|_i \leq \lambda$$

for some $\lambda > 0$. Use the simulated data and the function **`fmincon`** *in the estimation.*

Problem 13 *(continuation of previous problem) What happens to the parameters when n increases from 1000 to $10,000$ and to $100,000$?*

Problem 14 *(continuation of previous problem) What happens to the final squared-errors of the regression when λ increases?*

Problem 15 *(continuation of previous problem) (Optional) Consider the Lagrangian form of the problem:*

$$\min_{\beta_0,\beta_1,\ldots,\beta_{20}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_{i,1} - \ldots - \beta_{20}x_{i,20})^2 + \lambda\sum_{i=0}^{20}|\beta_i|$$

The constrained problem was written as an unconstrained problem, and the parameters can now be estimated with **`fminunc`**. *Estimate the parameters and repeat the analysis from the previous two problems.*

References

Arora, Rajesh Kumar (2015). *Optimization: algorithms and applications*. Chapman and Hall/CRC. URL: https://doi.org/10.1201/b18469.

Hayashi, F. (2000). *Econometrics*. Princeton University Press. ISBN: 9780691010182. URL: https://books.google.com/books?id=QyIW8WUIyzcC.