Computational Methods: Assignment 2

Matthew Mercuri

Introduction and Purpose

In this report, we will demonstrate computational methods that have far-reaching applications. The first section will focus on dynamic programming, specifically, an optimal stopping problem. The goal is to show mathematically how we can program a computer to stop participating in an activity once it reaches a critical point in relation to expected wealth. The next section will demonstrate a technique that can be used to find a local minimum for multi-dimensional functions. The last section will display how we can use the Gauss-Newton method for a common minimization problem.

Dynamic Programming - Optimal Stoppage Point

Question: Consider the following game. You are given a fair six-sided die, with faces {1, 2, 3, 4, 5, 6}. You can roll the die, and obtain a payoff equal to the value shown on the die, or you can choose to roll again. Note that if you choose to roll again, you can only get the new payoff or roll again. You can roll the die up to four times. What is your expected gain, assuming you follow the optimal strategy?

Although this game is simple, an optimal approach is non-trivial. A method would be considered "optimal" should it (on average) yield the maximum expected payoff. What follows is a mathematical approach to the game that will generate a well-defined expected gain. We define three formulas:

$$V_k = P_k \overline{W_k} + (1 - P_k) V_{k+1}$$
$$P_k = P(W_k > V_{k+1})$$
$$\overline{W_k} = E[W_k | W_k > V_{k+1}]$$

Where V_k is the expected payoff at *k* minus 1 throws remaining, P_k is the probability that the current payoff is greater than the expected payoff of continuing, and W_k is the expected value of a throw if it is greater than the expected wealth of continuing. To answer what is asked, we are essentially looking for V_4.

First, we have to calculate the expected value of the payoff if we were only to throw the die once:

$$V_1 = \frac{1}{6}(6 + 5 + 4 + 3 + 2 + 1) = \$3.50$$

From here, we can recursively apply the first formula to find V_4:

$$V_2 = \frac{1}{2}(\$5) + \frac{1}{2}(\$3.50) = \$2.50 + \$1.75 = \$4.25$$

$$V_3 = \frac{1}{3}(\$5.50) + \frac{2}{3}(\$4.25) = \$1.83 + \$2.83 = \$4.67$$

$$V_4 = \frac{1}{3}(\$5.50) + \frac{2}{3}(\$4.67) = \$1.83 + \$3.11 = \$4.94$$

So, the expected gain of this game is **$4.94**. In order to achieve this game (at least on average), you should always stop if you roll a 5 or a 6 given the expected gain. The probability of rolling a 5 or a 6 is:

$$P(Roll > 4) = 2\frac{1}{6} = \frac{1}{3}$$

Given this, it is most likely one should stop at 3 roles.

Optimality Condition - Local Minimum

Given the function:

$$f(x,y) = 3x^2y + x^2 - y^3 + 4xy^4 - 10$$

We are to verify whether the coordinate $x = 0$and $y = 0$is the local minimum of the function. To do this, we first compute the first-order partial derivatives:

$$f_x = 6xy + 2x + 4y^4$$

$$f_y = 3x^2 + 16xy^3 - 3y^2$$

For use later, we will also compute the second-order partial derivatives:

$$f_{xy} = f_{xy} = 6x + 16y^3$$

$$f_{xx} = 6y + 2$$

$$f_{yy} = 48xy^2 - 6y$$

Now using the gradient, we can find out if the point is a critical point in the function:

$$\nabla f(x,\ y) = (f_x,\ f_y) = (6xy + 2x + 4y^4,\ 3x^2 + 16xy^3 - 3y^2)$$

Plugging in $x = 0$and $y = 0$:

$$(6xy + 2x + 4y^4,\ 3x^2 + 16xy^3 - 3y^2) = (0,\ 0)$$

This immediately tells us that the given point is a critical point, though we do not yet know if it is a local minimum. For us to learn if it is, we can use a Hessian matrix employing the second-order partial derivatives we found earlier:

$$\nabla^2 f(x,\ y) = \begin{bmatrix} f_{xx} & f_{xy}; & f_{yx} & f_{yy} \end{bmatrix} = \begin{bmatrix} 6y + 2 & 6x + 16y^3; & 6x + 16y^3 & 48xy^2 - 6y \end{bmatrix}$$

$$\nabla^2 f(0,\ 0) = [2 \quad 0;\ 0 \quad 0]$$

Taking the determinant of the Hessian matrix, we get 0. This means our test for a minimum is inconclusive. We use a first-order test instead:

$$\lim_{x,\,y \to 0^{-}} f_x = 6xy + 2x + 4y^4 = 0$$

$$\lim_{x,\,y \to 0^{+}} f_x = 6xy + 2x + 4y^4 = 0$$

$$\lim_{x,\,y \to 0^{-}} f_y = 3x^2 + 16xy^3 - 3y^2 = 0$$

$$\lim_{x,\,y \to 0^{+}} f_y = 3x^2 + 16xy^3 - 3y^2 = 0$$

Given that our limits coming from both directions in the *xy* plane are zero, we can conclude that the coordinates are that of a local minimum.

Gauss-Newton Minimization

Solve the minimization problem

$$\min_x f(x), \tag{1}$$

where $x = [x_1, x_2]$, $f(x) \equiv \frac{1}{2} F^T(x) F(x)$ and $F(x) = \begin{bmatrix} 3x_1^2 + 2x_1x_2 - 1 \\ -3x_2 + 5x_1x_2 - 4 \\ e^{x_1} - \sin(x_2) + 1 \end{bmatrix}$, by the Gauss-Newton method without line search.

The first thing we can do is use a numerical approximation (finite difference method) to find the Jacobian matrix and evaluate the value *F*.

The general formula we use for the method in finding individual values within the Jacobian matrix is as follows:

$$J(i,\, j) = \frac{1}{\Delta x_j}\left[F(x + \Delta x_j e_j) - F(x)\right]$$

Of course, the Jacobian matrix's form is:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathrm{T}} f_1 \\ \vdots \\ \nabla^{\mathrm{T}} f_m \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}$$

https://wikimedia.org/api/rest_v1/media/math/render/svg/e343f872b676a0e64646f27593d03c77c53cbaf3

When looking for the value of *f(x)*, we want to find x such that its value is minimized. To do this, we follow the procedure:

$$A = J(x_k)^T \cdot J(x_k)$$

$$f = J(x_k)^T \cdot F(x_k)$$

$$Ah_{f_n} = -f$$

$$x_{k+1} = x_k + h_{f_n}$$

We apply this procedure continuously until we arrive at a value for *x* that minimizes *f*.

Part A

For the first part of this question, we are to implement (in Matlab) a function that allows us to evaluate *F(x)* using the finite difference method. The function should return *F* (the function value of *x*) and *J* for the Jacobian matrix.

The code that accomplishes this, looks like:

```
% Matthew Mercuri
% MTH 600 - Assignment 3
% Question 3 (a)

[F, J] = pfun([0 0]);
```

```
6.
7. function [F, J] = pfun(x)
8.
9.     % defining a small interval of x for numerical method
10.      delta_x = 0.01;
11.
12.      % creating placeholder matrices to store result
13.      J = zeros(3,2);
14.      j = zeros(3,2);
15.
16.      % function to output value of the first function when
   needed
17.      function f1_out = f1(f1_1, f1_2)
18.          f1_out = (3*(f1_1)^2)+(2*f1_1*f1_2)-1;
19.      end
20.
21.      % function to output value of the second function when
   needed
22.      function f2_out = f2(f2_1, f2_2)
23.          f2_out = (-3*f2_2)+(5*f2_1*f2_2)-4;
24.      end
25.
26.      % function to output value of the third function when
   needed
27.      function f3_out = f3(f3_1, f3_2)
28.          f3_out = exp(f3_1)-sin(f3_2)+1;
29.      end
30.
31.      % applying numerical method to find Jacobian matrix
32.      % basiclly, we increment one variable a small amount to
   find the
33.      % partial derivative, then evaluate it
34.      % first column
35.      x1_prime = x(1,1)+delta_x;
36.      x2 = x(1,2);
37.      J(1,1) = f1(x1_prime, x2);
38.      J(2,1) = f2(x1_prime, x2);
39.      J(3,1) = f3(x1_prime, x2);
40.
41.      % second column
42.      x1 = x(1,1);
43.      x2_prime = x(1,2)+delta_x;
44.      J(1,2) = f1(x1, x2_prime);
45.      J(2,2) = f2(x1, x2_prime);
46.      J(3,2) = f3(x1, x2_prime);
```

```
47.
48.      % finding matrix with unaltered input values for Jacobian
   approximation
49.      j(1,1) = f1(x1, x2);
50.      j(2,1) = f2(x1, x2);
51.      j(3,1) = f3(x1, x2);
52.      j(1,2) = f1(x1, x2);
53.      j(2,2) = f2(x1, x2);
54.      j(3,2) = f3(x1, x2);
55.
56.      % applying numerical method (finite difference)
57.      J = (J-j)*(1/delta_x);
58.      F = j(:,1); % storing value of F
59.
60.      % displaying results
61.      disp(J)
62.      disp(F)
63.   end
```

Part B

The second part, has us using the Gauss-Newton method without line search to find the value of x that minimizes the objective function and calculates the value for it. The function that achieves this, using the function from above. The code is well documented so it explains what it is doing.

```
1. % Matthew Mercuri
2. % MTH 600 - Assignment 3
3. % Question 3 (b)
4.
5. % Gauss-Newton method for solving nonlinear least squares
   problem without line search
6. %
7. % Input
8. % fun - function F(x), the objective function is 1/2*F'*F,
   which returns
9. % F(x) and corresponding Jacobian matrix J.
10.   % x0 - initial value of x
11.   % itmax - max number of iteration
12.   % tol - stopping tolerance
13.   %
```

```
14.   % Output
15.   % x - final result
16.   % it - number of iterations
17.   % r - objective function value 1/2*F'*F
18.
19.   Gauss_Newton([1; 1], 100, 1e-4);
20.
21.   function [x, it, r] = Gauss_Newton(x0, itmax, tol)
22.       r = zeros(1, itmax); % creating vector to store values for
      objective function
23.       [F, J] = pfun(x0); % find the value for F and the Jacobian
      matrix
24.       A = transpose(J)*J;
25.       f = transpose(J)*F;
26.       hf = A\(-f); % to update our current guess of x
27.       x = x0+hf; % updating current guess of x
28.       r(1,1) = 0.5*dot(F, transpose(F)); % calculating value for
      objective function
29.       for i = 2:itmax
30.           [F, J] = pfun(x);
31.           A = transpose(J)*J;
32.           f = transpose(J)*F;
33.           hf = A\(-f);
34.           x = x+hf;
35.
36.           r(1,i) = 0.5*dot(F, transpose(F));
37.
38.           % checking to see if we are within our given tolerance
39.           if abs((r(1, i)-r(1, i-1))) < tol
40.               r = r(1, 1:i);
41.               it = i;
42.               break
43.           end
44.       end
45.       disp(r)
46.       disp(x)
47.       disp(it)
48.
49.       % plotting the convergence
50.       plot(1:it, r)
51.       title('r convergence')
52.       xlabel('iteration')
53.       ylabel('value of r')
54.
55.   end
```

```
56.
57.
58.   function [F, J] = pfun(x)
59.
60.       % defining a small interval of x for numerical method
61.       delta_x = 0.01;
62.
63.       % creating placeholder matrices to store result
64.       J = zeros(3,2);
65.       j = zeros(3,2);
66.
67.       % function to output value of the first function when
    needed
68.       function f1_out = f1(f1_1, f1_2)
69.           f1_out = (3*(f1_1)^2)+(2*f1_1*f1_2)-1;
70.       end
71.
72.       % function to output value of the second function when
    needed
73.       function f2_out = f2(f2_1, f2_2)
74.           f2_out = (-3*f2_2)+(5*f2_1*f2_2)-4;
75.       end
76.
77.       % function to output value of the third function when
    needed
78.       function f3_out = f3(f3_1, f3_2)
79.           f3_out = exp(f3_1)-sin(f3_2)+1;
80.       end
81.
82.       % applying numerical method to find Jacobian matrix
83.       % basiclly, we increment one variable a small amount to
    find the
84.       % partial derivative, then evaluate it
85.       % first column
86.       x1_prime = x(1,1)+delta_x;
87.       x2 = x(2,1);
88.       J(1,1) = f1(x1_prime, x2);
89.       J(2,1) = f2(x1_prime, x2);
90.       J(3,1) = f3(x1_prime, x2);
91.
92.       % second column
93.       x1 = x(1,1);
94.       x2_prime = x(2,1)+delta_x;
95.       J(1,2) = f1(x1, x2_prime);
96.       J(2,2) = f2(x1, x2_prime);
```

```
97.      J(3,2) = f3(x1, x2_prime);
98.
99.      % finding matrix with unaltered input values for Jacobian
   approximation
100.     j(1,1) = f1(x1, x2);
101.     j(2,1) = f2(x1, x2);
102.     j(3,1) = f3(x1, x2);
103.     j(1,2) = f1(x1, x2);
104.     j(2,2) = f2(x1, x2);
105.     j(3,2) = f3(x1, x2);
106.
107.     % applying numerical method (finite difference)
108.     J = (J-j)*(1/delta_x);
109.     F = j(:,1); % storing value of F
110.  end
```

Part C

To use the code, we specify two inputs:

1. x0=[1;1], tol=1e-4, itmax=100
2. x0=[0;0], tol=1e-5, itmax=100

The results are summarized in the following table:

1	x = [-0.5443, -0.6263]	r(x) = 2.5963	Iterations = 9
2	x = [-0.5446, -0.6261]	r(x) = 2.5962	Iterations = 9

From the table, it is clear that both converge to a solution, hence the slight differences. Although the second input has a lower tolerance, the solution is found just as quickly. This is likely because the guess for x0 is better. Normally, we would expect that there would be more iterations done as a tradeoff for a more accurate result.

Part D

We can graph the simulation, to obtain a visual representation of the convergence. Below is a figure that shows our calculated value for *r(x)* compared to the iteration. Of course, we expect it to quickly converge towards a specific value. This value is the minimization that we sought from above.