Steepest Descent

A Minimization Problem

Jordan Saethre

December 12, 2019

Contents

1 Introduction **3**
1.1 Computed Example 3
1.2 Steepest Descent 3

2 Single Variable Function **4**
2.1 Python Code for Single Variable Case 4

3 Two Variable Functions **6**
3.1 Python Code for Two Variable Case 7

4 Original Problem **8**
4.1 Improvements 8
4.2 Python Code for Original Problem 9

1 Introduction

Engineers and statisticians alike are often curious about how much information can be obtained directly from the resulting data of an experiment. It is up to the mathematician to analyze the methods used to determine the validity of inferences made by such individuals. This was, in short, the burden of the article "Some technical thoughts on modeling", [1]. It was desired to find out in particular if the parameters of a differential equation can be obtained from the data of an experiment. It was even more interesting to discover just how little data need be used to sequester the parameters of interest.

The intention of this project was to replicate what was done in [1] and to explore the challenges surrounding this basic algorithm in hopes of laying a path forward for improvement.

1.1 Computed Example

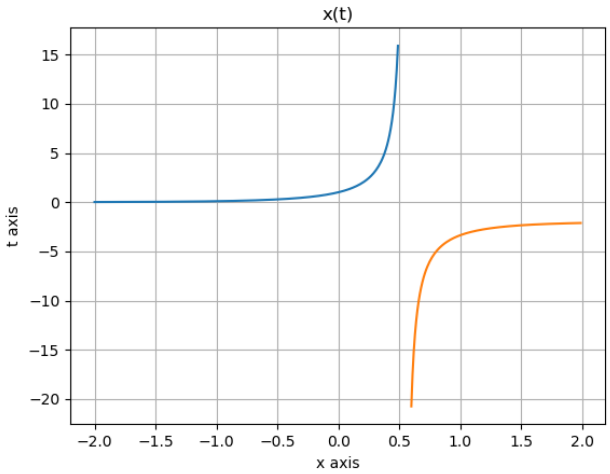
To test this algorithm the solution of a differential equation is computed.

$$x'(t) = x(t)^2 + 2x(t); \quad x(0) = 1$$

which has known solution

$$x(t) = \frac{-2e^{2t}}{e^{2t} - 3}$$

Keeping in mind possible singularities in this solution, data is collected over an appropriate domain, say on the interval [1,2].



n points are generated on this interval and are used to minimize the value of the following function:

$$F(a_1, a_2) = \sum_{i=1}^{n} \left(a_1 x(t_i)^2 + a_2 x(t_i) - \frac{x(t_{i+1}) - x(t_i)}{t_{i+1} - t_i} \right)^2$$

Since the parameters of this differential equation are already known it is hoped that the algorithm will return $a_1 = 1$ and $a_2 = 2$. And if it does, it would appear that this algorithm could be trusted to ascertain parameters of a less defined system describing some phenomena of interest.

1.2 Steepest Descent

The idea behind this method is simple. Begin with a guess for the parameters that minimizes the function $F(a_1, a_2)$. The value of the function along with the gradient (∇F) is calculated from this initial guess. The gradient evaluated at a particular site points in the direction of steepest ascent which implies that $-\nabla F|_{(a_1,a_2)}$ points in the direction of greatest *descent*.

For this particular problem

$$\nabla F|_{(a_1,a_2)} = \begin{bmatrix} \frac{\partial}{\partial a_1}(F(a_1,a_2)) \\ \\ \frac{\partial}{\partial a_2}(F(a_1,a_2)) \end{bmatrix}^T = \begin{bmatrix} \sum_{i=1}^{n} 2x(t_i)^2 \left(a_1 x(t_i)^2 + a_2 x(t_i) - \frac{x(t_{i+1})-x(t_i)}{t_{i+1}-t_i}\right) \\ \\ \sum_{i=1}^{n} 2x(t_i) \left(a_1 x(t_i)^2 + a_2 x(t_i) - \frac{x(t_{i+1})-x(t_i)}{t_{i+1}-t_i}\right) \end{bmatrix}^T$$

This is used to edit the initial guess as follows:

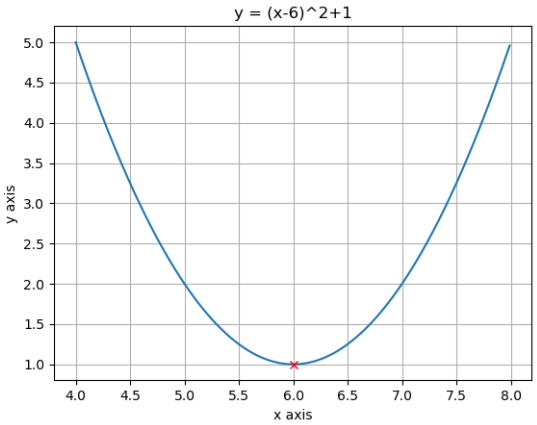
Let (a_{1i}, a_{2i}) be the i^{th} guess for parameters (a_1, a_2)

$$(a_{1i}, a_{2i}) = (a_{1(i-1)}, a_{2(i-1)}) - \nabla F|_{a_{1(i-1)}, a_{2(i-1)}} \cdot \epsilon$$

$F(a_{1i}, a_{2i})$ is then compared to $F(a_{1(i-1)}, a_{2(i-1)})$. If $F(a_{1(i-1)}, a_{2(i-1)}) < F(a_{1i}, a_{2i})$ the process is repeated. If ever the next guess is greater, a smaller epsilon is used until a minimum is found within the desired precision.

2 Single Variable Function

Before attacking the main problem the process is abstracted and simplified starting with the single variable case. The idea here is to write code to find the minimum of a very simple single variable function. Once this milestone has been reached the code can be adapted to handle the 2 variable case in which the minimum of a surface is sought. Consider the the following curve $y = (x-6)^2 + 1$ with minimum at $(6, 1)$.



2.1 Python Code for Single Variable Case

The code takes in four inputs: maximum number of iterates, delta (initial step size), epsilon (desired precision), and an initial guess for the x value of the minimum. With the following inputs the result was calculated to be $(5.996039, 1.000016)$.

- iterates: 1000
- delta: 0.1
- epsilon: 0.001
- initial guess: 2

Code for Single Variable Minimum

```
import numpy as np

# function
my_function = lambda x: (x-6)**2 + 1

# derivative of function
dfdx = lambda x: 2*(x-6)

# find minimum
# interates: max number of times it will iterate
# delta: beginning step size
# epsilon: percision
# x_i: initial guess
def find_min(iterates, delta, epsilon, x_i):
    n = 0
    f_i = my_function(x_i)
    for n in range(0, iterates):
        df_i = dfdx(x_i)
        x_i_n = x_i - df_i*delta
        f_i_n = my_function(x_i_n)
        if (f_i_n < f_i):
            if (np.abs(x_i - x_i_n) <= epsilon):
                print(x_i_n, my_function(x_i_n))
                break
                return my_function(x_i_n)
            x_i = x_i_n
            f_i = f_i_n
            n = n + 1
        else:
            delta = delta/2
            n = n + 1

find_min(1000, 0.1, 0.001, 2)
```

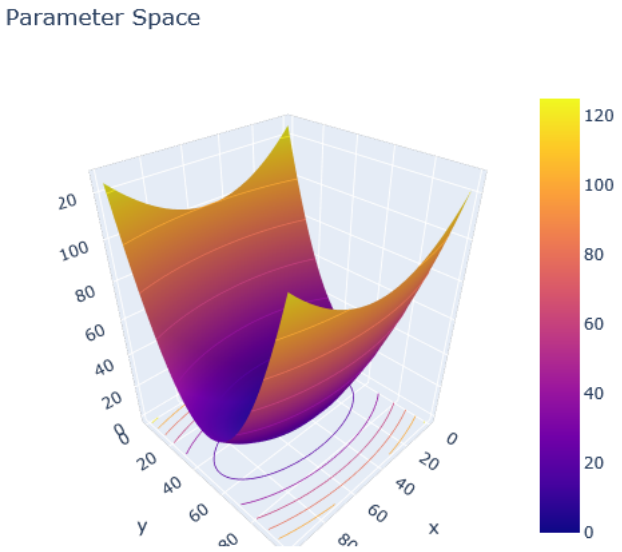
3 Two Variable Functions

To adapt the code to handle the two variable case a surface with one minimum is chosen as a test. The following paraboloid has one minimum at $(2, 3)$.

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

The following inputs were able to approximate the minimum to be at $(2.000000, 2.996441)$.

- iterates: 1000
- delta: 0.1
- epsilon: 0.001
- initial guess: $(1.3, 0.7)$

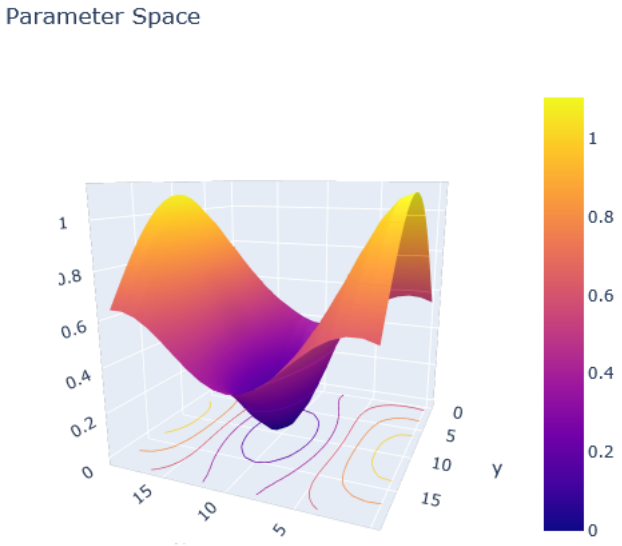


A slightly more complicated surface was chosen to test the code on. This surface has global maximum at $(3, 1)$.

$$f(x, y) = ((x-3)^2 + 3(y-1)^2)e^{-(x-3)^2-(y-1)^2}$$

The following inputs were able to approximate the minimum to be at $(2.996786, 0.999999)$.

- iterates: 1000
- delta: 0.1
- epsilon: 0.001
- initial guess: $(2.8, 0.8)$



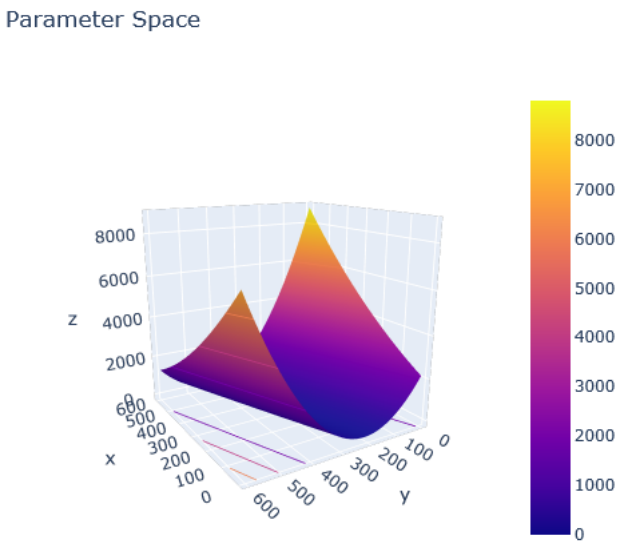
This surface did suffer this issue surrounding initial guesses. The code would return local minimum if the initial guess was too far off from the point that was intended to be found. Possible fixes for this issue is discussed in section 4.

3.1 Python Code for Two Variable Case

```
import numpy as np

# example A: function with minimum at (2,3)
my_function = lambda x,y: 4*(x-2)**2+(y-3)**2
# example B: function with minimum at (3,1)
# my_function = lambda x,y: ((x-3)**2
                # + 3*(y-1)**2)*np.exp(-(x-3)**2-(y-1)**2)

# example A: partial derivative of f with respect to x
fx = lambda x,y: 8*(x-2)
# example B: partial derivative of f with respect to x
# fx = lambda x,y: ((x-3)**2
       # + 3*(y-1)**2 - 1)*np.exp(-(x-3)**2-(y-1)**2)*(-2*(x-3))

# example A: partial derivative of f with respect to y
fy = lambda x,y: 2*(y-3)
# example B: partial derivative of f with respect to y
# fy = lambda x,y: ((x-3)**2
    # + 3*(y-1)**2 - 1)*np.exp(-(x-3)**2-(y-1)**2)*(-6*(y-1))

# example A: initial guess for minimum
xy_i = np.array([1.3, .7])
# example B: initial guess for minimum
# xy_i = np.array([2.8, .8])

def find_min(iterates, delta, epsilon, xy_i):
    n = 0
    f_i = my_function(xy_i[0], xy_i[1])
    for n in range(0, iterates):
        df_i = np.array([fx(xy_i[0],xy_i[1]),fy(xy_i[0], xy_i[1])])
        xy_i_n = xy_i - df_i*delta
        f_i_n = my_function(xy_i_n[0], xy_i_n[1])
        print(df_i, xy_i_n, f_i_n)
        if (f_i_n <= f_i):
            if (np.abs(xy_i[0] - xy_i_n[0]) <= epsilon and
                    np.abs(xy_i[1] - xy_i_n[1]) <= epsilon ):
                print(xy_i_n, my_function(xy_i_n[0], xy_i_n[1]))
                break
                return my_function(xy_i_n[0], xy_i_n[1])
            xy_i = xy_i_n
            f_i = f_i_n
            n = n + 1
        else:
            delta = delta/2
            n = n + 1
find_min(1000, 0.1, 0.001, xy_i)
```

4 Original Problem

To tackle the main problem some additional code blocks needed to be created. Code was written to gather the data points from the actual solution. The gradient was calculated by hand on paper and then a code was written to carry out the calculations. The code for calculating $F(a_1, a_2)$, the height of the surface at any point (a_1, a_2) was written. The two variable minimum finding code described above was then edited to account for these new elements.



To the right is an image of the surface of which the minimum is sought. The x-axis is for possible a_1 values and the y-axis is for a_2 values. While it appears too look like a sheet of paper with the corners pulled up there is in fact a very slight curvature along the bottom, that is, there is a global minimum at $(1, 2)$. This is exactly what is expected based on the original differential equation. The code takes in all of the same inputs as before with the addition of n points to be collected between (a, b).

With the following inputs the code returned an approximated minimum at $(1.063961, 2.273921)$.

- iterates: 1000
- delta: 0.1
- epsilon: 0.0001
- n = 10
- (a,b) = (1, 2)
- initial guess: (2, 2.5)

4.1 Improvements

The minimums found for the test cases were much more accurate than what the code for the main problem yielded. This may be due to the fact that the gradient becomes extremely shallow once close enough to the true minimum. It should also be noted that only 10 points were used in the minimization problem and that increasing this number of points would likely improve the accuracy but at the cost of efficiency. Using more points would require many more calculations. While the code ran and returned values for all examples relatively instantaneously, this could become a larger concern if using large data sets from experiments.

The issue encountered in section 3 with a surface involving more than one minimum could be addressed by writing a new loop that would cycle through many possible initial guesses and return the most minimal point out of all the guesses. This way if the code converges to a local minimum the chances are higher that the global minimum will actually be found.

4.2 Python Code for Original Problem

```
import numpy as np
from matplotlib import pyplot as plt

def get_data(a,b,n): # get n x_i's and t_i's between a and b
    my_function = lambda x: (-2*np.exp(2*x))/(np.exp(2*x)-3)
    x_t = np.array([])
    t_i = np.array([])
    for i in range(0, n + 1):
        step = np.abs((b-a)/n)
        x_t = np.append(x_t, my_function(a + i*step))
        t_i = np.append(t_i, a + i*step)
        i = i + 1
    return x_t, t_i


# Calculate F_a given [a_1, a_2]
def F_a(a_i, x_t_data, t_data):
    sum_F_a = 0
    for i in range(0, len(x_t_data) - 1):
        sum_F_a = sum_F_a + np.abs(a_i[0]*(x_t_data[i])**2
            + a_i[1]*(x_t_data[i]) - ((x_t_data[i + 1]
            - x_t_data[i])/(t_data[i + 1] - t_data[i])))**2
    return sum_F_a


# Calculate delF given [a_1, a_2]
def del_F_a(a_i, x_t_data, t_data):
    sum_del_F_a = np.array([0.,0.])
    for i in range(0, len(x_t_data) - 1):
        sum_del_F_a[0] = sum_del_F_a[0] + 2*x_t_data[i]**2*
                        (a_i[0]*(x_t_data[i])**2
                            + a_i[1]*(x_t_data[i])
                            - ((x_t_data[i + 1]
                            - x_t_data[i])/(t_data[i + 1]
                            - t_data[i])))
        sum_del_F_a[1] = sum_del_F_a[1] + 2*x_t_data[i]*
                        (a_i[0]*(x_t_data[i])**2
                            + a_i[1]*(x_t_data[i])
                            - ((x_t_data[i + 1]
                            - x_t_data[i])/(t_data[i + 1]
                            - t_data[i])))
    return sum_del_F_a


# find minimum
# inputs:
```

```
# # interates: max number of times it will iterate
# # delta: beginning step size
# # epsilon: percision
# # a_i: initial guess
# # (a,b): interval to get data from
# # n: number of points to get between a and b
def find_min(iterates, delta, epsilon, a_i, a, b, n):
    k = 0
    x_t, t_i = get_data(a,b,n)[0], get_data(a, b, n)[1]
    f_i = F_a(a_i, x_t, t_i)
    print(f_i)
    for k in range(0, iterates):
        df_i = del_F_a(a_i, x_t, t_i)
        a_i_n = a_i - df_i*delta*
                (1/((df_i[0]**2 + df_i[1]**2)**(1/2)))
                # last term is to make it a unit vector
        f_i_n = F_a(a_i, x_t, t_i)
        print(delta)
        print(a_i_n)
        print(f_i_n)
        if (f_i_n <= f_i):
            if (np.abs(a_i[0] - a_i_n[0]) <= epsilon and
                np.abs(a_i[1] - a_i_n[1]) <= epsilon ):
                print(a_i_n, F_a(a_i_n, x_t, t_i))
                break
                return a_i_n, F_a(a_i_n, x_t, t_i)
            a_i = a_i_n
            f_i = f_i_n
            k = k + 1
        else:
            delta = delta/2
            k = k + 1


# inputs
iterates = 100
delta = 0.1
epsilon = 0.0001
a_i = np.array([.8, 3])
a = 3
b = 4
n = 100

# function call
find_min(iterates, delta, epsilon, a_i, a, b, n)
```

References

[1] N. O. Aksamit, D. H. Tucker, and J. F. Tucker, "Some technical thoughts on modeling," 2015.