2.6: Unconstrained Optimization- Numerical Methods

The types of problems that we solved in the previous section were examples of unconstrained optimization problems. That is, we tried to find local (and perhaps even global) maximum and minimum points of real-valued functions \( f(x, y) \), where the points \((x, y)\) could be any points in the domain of \( f \). The method we used required us to find the critical points of \( f \), which meant having to solve the equation \( \nabla f = \mathbf{0} \), which in general is a system of two equations in two unknowns (\( x, y \)). While this was relatively simple for the examples we did, in general this will not be the case. If the equations involve polynomials in \( x, y \) of degree three or higher, or complicated expressions involving trigonometric, exponential, or logarithmic functions, then solving even one such equation, let alone two, could be impossible by elementary means.

For example, if one of the equations that had to be solved was

\[
x^3 + 9x - 2 = 0
\]

you may have a hard time getting the exact solutions. Trial and error would not help much, especially since the only real solution turns out to be

\[
\sqrt[3]{\sqrt{28} + 1 - \sqrt[3]{\sqrt{28} - 1}}
\]

In a situation such as this, the only choice may be to find a solution using some numerical method which gives a sequence of numbers which converge to the actual solution. For example, Newton’s method for solving equations \( f(x) = 0 \), which you probably learned in single-variable calculus. In this section we will describe another method of Newton for finding critical points of real-valued functions of two variables.

Let \( f(x, y) \) be a smooth real-valued function, and define
\[
D(x, y) = \frac{\partial^2 f}{\partial x^2} (x, y) \frac{\partial^2 f}{\partial y^2} (x, y) - \left(\frac{\partial^2 f}{\partial y \partial x} (x, y)\right)^2
\]

**Newton's algorithm**: Pick an initial point \((x_0, y_0)\). For \((n) = 0, 1, 2, \ldots\), define:

\[
x_{n+1} = x_n - \frac{\begin{vmatrix}
\frac{\partial^2 f}{\partial y^2} (x_n, y_n) & \frac{\partial^2 f}{\partial x \partial y} (x_n, y_n) \\
\frac{\partial f}{\partial y} (x_n, y_n) & \frac{\partial f}{\partial x} (x_n, y_n)
\end{vmatrix}}{D(x_n, y_n)},
\quad
y_{n+1} = y_n - \frac{\begin{vmatrix}
\frac{\partial^2 f}{\partial x^2} (x_n, y_n) & \frac{\partial^2 f}{\partial x \partial y} (x_n, y_n) \\
\frac{\partial f}{\partial x} (x_n, y_n) & \frac{\partial f}{\partial y} (x_n, y_n)
\end{vmatrix}}{D(x_n, y_n)}
\]

Then the sequence of points \((x_n, y_n) \}_{n=1}^{\infty}\) converges to a critical point. If there are several critical points, then you will have to try different initial points to find them.

Example 2.23

Find all local maxima and minima of \(f(x, y) = x^3 - x y - x + x y^3 - y^4\).

**Solution**

First calculate the necessary partial derivatives:

\[
\frac{\partial f}{\partial x} = 3x^2 - y - 1 + y^3, \quad \frac{\partial f}{\partial y} = -x + 3x y^2 - 4y^3\]

\[
\frac{\partial^2 f}{\partial x^2} = 6x, \quad \frac{\partial^2 f}{\partial y^2} = 6x y - 12y^2, \quad \frac{\partial^2 f}{\partial y \partial x} = -1 + 3y^2
\]

Notice that solving \(\nabla f = \textbf{0}\) would involve solving two third-degree polynomial equations in \(x \text{ and } y\), which in this case cannot be done easily.

We need to pick an initial point \((x_0, y_0)\) for our algorithm. Looking at the graph of \(z = f(x, y)\) over a large region may help (see Figure 2.6.1 below), though it may be hard to tell where the critical points are.

![Figure 2.6.1](image.png)

\(f(x, y) = x^3 - x y - x + x y^3 - y^4\) for \(-20 \leq x \leq 20\) and \(-20 \leq y \leq 20\)
Notice in the Formulas Equation \ref{Eq2.14} that we divide by \(D\), so we should pick an initial point where \(\sqrt{D}\) is not zero. And we can see that \(\sqrt{D(0,0)} = \sqrt{(0)(0)−(−1)^2} = -1 \neq 0\), so take \((0,0)\) as our initial point. Since it may take a large number of iterations of Newton’s algorithm to be sure that we are close enough to the actual critical point, and since the computations are quite tedious, we will let a computer do the computing. For this, we will write a simple program, using the Java programming language, which will take a given initial point as a parameter and then perform 100 iterations of Newton’s algorithm. In each iteration the new point will be printed, so that we can see if there is convergence. The full code is shown in Listing 2.1.

```java
public class newton {
    public static void main(String[] args) {
        // Set the initial point (x,y) as command-line parameters
        double x = Double.parseDouble(args[0]); // Initial x value
        double y = Double.parseDouble(args[1]); // Initial y value
        System.out.println("Initial point: "; x*" * " + y*" * "");
        // Go through 100 iterations of Newton’s algorithm
        for (int n=1; n<=100; n++) {
            double D = f(x,y)+f(y,x) - Math.pow(fy(x,y),2);
            double xn = x; double yn = y; // The current x and y values
            if (D == 0) { // We can not divide by 0
                System.out.println("Error: D = 0 at iteration n = " + n);
                System.exit(0); // End the program
            } else { // Calculate the new values for x and y
                x = xn - (fyy(xn,yn)+fxx(xn,yn)-fxy(xn,yn)*fy(xn,yn))/D;
                y = yn - (fxx(xn,yn)+fyy(xn,yn)-fxy(xn,yn)*fx(xn,yn))/D;
                System.out.println("n = " + n + "; "; x + " * " + y + " * ");
            }
        }
        // Below are the parts specific to the function
        // The first partial derivative of f wrt x: 3x^2-2xy+y^3
        public static double f(double x, double y) {
            return 3*x*Math.pow(x,2) - y - 1 + Math.pow(y,3);
        }
        // The first partial derivative of f wrt y: -x^3+3y^2-4y^3
        public static double fy(double x, double y) {
            return -x + 3*x*Math.pow(y,2) - 4*Math.pow(y,3);
        }
        // The second partial derivative of f wrt x: 6x
        public static double fxx(double x, double y) {
            return 6*x;
        }
        // The second partial derivative of f wrt y: 6y-12y^2
        public static double fyy(double x, double y) {
            return 6*x*y - 12*y*Math.pow(y,2);
        }
        // The mixed second partial derivative of f wrt x and y: -12y^2
        public static double fxy(double x, double y) {
            return -1 + 3*Math.pow(y,2);
        }
    }
}
```

Listing 2.1 Program listing for newton.java

To use this program, you should first save the code in Listing 2.1 in a plain text file called newton.java. You will need the Java Development Kit to compile the code. In the directory where newton.java is saved, run this command at a command prompt to compile the code: `javac newton.java`

Then run the program with the initial point (0,0) with this command:

`java newton 0 0`

Below is the output of the program using (0,0) as the initial point, truncated to show the first 10 lines and the last 5 lines:
As you can see, we appear to have converged fairly quickly (after only 8 iterations) to what appears to be an actual critical point (up to Java’s level of precision), namely the point (0.47113563449874, -0.39636433796318005). It is easy to confirm that \( f = 0 \) at this point, either by evaluating \( \frac{\partial f}{\partial x} \) and \( \frac{\partial f}{\partial y} \) at the point ourselves or by modifying our program to also print the values of the partial derivatives at the point. It turns out that both partial derivatives are indeed close enough to zero to be considered zero:

\[
\frac{\partial f}{\partial x} (0.47113563449874, -0.39636433796318005) = 4.85722573273506 \times 10^{-17}
\]

\[
\frac{\partial f}{\partial y} (0.47113563449874, -0.39636433796318005) = -8.326672684688674 \times 10^{-17}
\]

We also have \( D(0.47113563449874, -0.39636433796318005) = -8.77607563032301 < 0 \), so by Theorem 2.6 we know that (0.47113563449874, -0.39636433796318005) is a saddle point.

Since \( f \) consists of cubic polynomials, it seems likely that there may be three critical points. The computer program makes experimenting with other initial points easy, and trying different values does indeed lead to different sequences which converge:
Again, it is easy to confirm that both $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ vanish at the point $(-0.6703832459238667,0.42501465652420045)$, which means it is a critical point. And

$D(-0.6703832459238667,0.42501465652420045) = 15.3853578526055 > 0$

$\frac{\partial^2 f}{\partial x^2} (-0.6703832459238667,0.42501465652420045) = -4.0222994755432 < 0$

so we know that $(-0.6703832459238667,0.42501465652420045)$ is a local maximum. An idea of what the graph of $f$ looks like near that point is shown in Figure 2.6.2, which does suggest a local maximum around that point.

Finally, running the computer program with the initial point $(-5,-5)$ yields the critical point $(-7.540962756992551,-5.595509445899435)$, $D < 0$ at that point, which makes it a saddle point.
The derivation of Newton’s algorithm, and the proof that it converges (given a “reasonable” choice for the initial point) requires techniques beyond the scope of this text. See RALSTON and RABINOWITZ for more detail and for discussion of other numerical methods. Our description of Newton’s algorithm is the special two-variable case of a more general algorithm that can be applied to functions of \(n \geq 2\) variables.

In the case of functions which have a global maximum or minimum, Newton’s algorithm can be used to find those points. In general, global maxima and minima tend to be more interesting than local versions, at least in practical applications. A maximization problem can always be turned into a minimization problem (why?), so a large number of methods have been developed to find the global minimum of functions of any number of variables. This field of study is called nonlinear programming. Many of these methods are based on the steepest descent technique, which is based on an idea that we discussed in Section 2.4. Recall that the negative gradient \(\langle - \nabla f \rangle\) gives the direction of the fastest rate of decrease of a function \(\langle f \rangle\). The crux of the steepest descent idea, then, is that starting from some initial point, you move a certain amount in the direction of \(\langle - \nabla f \rangle\) at that point. Wherever that takes you becomes your new point, and you then just keep repeating that procedure until eventually (hopefully) you reach the point where \(f\) has its smallest value. There is a “pure” steepest descent method, and a multitude of variations on it that improve the rate of convergence, ease of calculation, etc. In fact, Newton’s algorithm can be interpreted as a modified steepest descent method. For more discussion of this, and of nonlinear programming in general, see BAZARAA, SHERALI and SHETTY.

Contributors

- Michael Corral (Schoolcraft College). The content of this page is distributed under the terms of the GNU Free Documentation License, Version 1.2.