Numerical Methods for Vector Calculus

Ian Hollas - Honors Vector Calculus - Final Project

I implemented several numerical algorithms for solving problems involving vector calculus. Specifically, I built a program to approximate derivatives using first differences. I used this to implement gradient descent, an algorithm for finding local minima numerically. I also coded three numerical integration techniques (Monte Carlo, trapezoid rule, and Clenshaw-Curtis) and built a program to approximate path integrals with them.

The code (written in Python) can be found in a GitHub repository here: https://github.com/imhollas/vector-calc. This report includes a description and analysis of the code.

Contents

1 Finite Differences 3

2 Gradient Descent 3

3 Numerical Integration Techniques 4

4 Path Integrals 5

5 Sources 5

1 Finite Differences

Given a smooth function $f\colon \mathbf{R} \to \mathbf{R}$, we wish to efficiently approximate its derivative at a point x_0.
The basic solution is to use the definition of the derivative:

$$f'(x_0) = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}.$$

This naturally implies an approximation:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}.$$

How good is this? We can use a Taylor series to find how the error grows with Δx:

$$\begin{aligned}\epsilon &= \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} - f'(x_0) \\ &= \frac{f'(x_0)\Delta x + f''(x_0)(\Delta x)^2/2 + O((\Delta x)^3)}{\Delta x} - f'(x_0) \\ &= f''(x_0)\Delta x/2 + O((\Delta x)^2).\end{aligned}$$

So the error for this approximation is linear with the step size. Surprisingly, we can do better for free by taking a "centered difference" instead of a "forward difference":

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x}.$$

The f'' terms in cancel, so the error is quadratic in the step size:

$$\begin{aligned}\epsilon &= \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} - f'(x_0) \\ &= \frac{[f''(x_0)(\Delta x)^2/2 + O((\Delta x)^3)] - [f''(x_0)(-\Delta x)^2/2 + O((\Delta x)^3)]}{2\Delta x} \\ &= O((\Delta x)^2).\end{aligned}$$

I used this idea (which I found in Strang, see sources) in my program (gradient.py) that numerically evaluates the gradient of any function at a given point. The program takes a function, a position, and a step size and outputs the vector of partial derivatives. One could easily apply use this to compute other things of interest (Jacobian, divergence, curl). The gradient, in particular, has an interesting application to minimization problems.

2 Gradient Descent

The problem: given a smooth function $f\colon D \to \mathbf{R}$ with $D \subseteq \mathbf{R}^n$, find local minima (or maxima) of f on D numerically.

I implemented a basic version of an algorithm called gradient descent. The idea is to start at a random point in D, compute the gradient ∇f at that point (analytically or numerically), and then take a step $\alpha \nabla f$, where α is a constant. For minima, choose $\alpha < 0$, and for maxima, choose $\alpha > 0$. This process is repeated until $|\nabla f| < \epsilon$ for some small ϵ (my code uses $\epsilon = 10^{-5}$), or $N \gg 1$ steps occur (to ensure the program terminates; I used $N = 10^7$). The program returns the current position and the number of steps taken.

For code testing, I used a slight modification of problem 3 from homework 4: minimize $f(x, y) = xy + 1/x + 1/y$ on the first quadrant, restricted to the domain $(0, 100) \times (0, 100)$ (a finite domain is necessary to be able to choose a random point). Analytically, we can show that this function has a local minima at $(1, 1)$. I ran 1000 trials of choosing a random point in the domain and running the algorithm described above, with $\alpha = -1$. The process jumped out of the desired region 619 times, so we ignore those cases. In the remaining cases, it converged to $(1, 1)$ within four decimal points of accuracy. It took an average of ≈ 9200 steps, with standard deviation ≈ 3600.

3 Numerical Integration Techniques

The basic problem: given a function $f\colon [a, b] \to \mathbf{R}$, efficiently approximate the integral I of that function over the interval. Analytic techniques (setting up bounds) can be used in conjunction with the methods here to integrate functions on domains in $\mathbf{R}^n$.

The first method that I used was the "trapezoid rule." The idea is to divide the interval into N even subintervals. We can approximate the integral I_i over the kth subinterval by

$$I_i \approx \frac{f(a + (k-1)\Delta x) + f((a+k)\Delta x)}{2}\Delta x,$$

where $\Delta x = (b - a)/N$. In words, we are taking a rough average of the function on the interval. Geometrically, this is the area of a trapezoid, hence the name. The integral is thus approximated by

$$\begin{aligned} I &\approx \sum_{k=1}^{N} \frac{f(a + (k-1)\Delta x) + f((a+k)\Delta x)}{2}\Delta x \\ &= \Delta x \left[\frac{f(a) + f(b)}{2} + \sum_{k=1}^{N-1} f(a + k\Delta x)\right]. \end{aligned}$$

This is also known as the Euler-Maclaurin formula. My program integration.py implements this and my line integral program currently uses it to compute integrals.

The second technique I implemented is a version of the Monte Carlo technique. The basic idea is that you pick a random point $t_i \in [a, b]$, evaluate $f(t_i)$, repeat this $N \gg 1$ times, and average to get

$$I \approx \frac{1}{N}\sum_{i=1}^{N} f(t_i).$$

The point choosing process can be described by a probability distribution function $p\colon [a, b] \to \mathbf{R}$ defined by $p(x) = 1/(b - a)$. The expected value of $f(x)$ is therefore

$$\langle f(x)\rangle = \int_a^b f(x)p(x)dx = \frac{1}{b-a}\int_a^b f(x)dx = \frac{I}{b-a}.$$

By the law of large numbers, the random process of choosing t_i, evaluating f, and averaging will converge to $\langle f(x)\rangle$ as $N \to \infty$, so this method will work for approximating I.

4 Path Integrals

5 Sources