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, trapezoids, 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. I opted to implement as much of the math myself as possible, as opposed to using external libraries. This report includes a description and analysis of the code.

Contents

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 problem 3 from homework 4: minimize $f(x, y) = xy + 1/x + 1/y$ on the first quadrant. Further, I restricted to the domain $(0, 100) \times (0, 100)$, as a finite domain is necessary to choose a random point. Analytically, we can show that this function has a local minima at $(1, 1)$, making it a good test case. 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 (final position outside), so we ignore those cases. In the remaining cases, it converged to $(1, 1)$ within 10^{-4}. It took an average of ~ 9200 steps, with standard deviation ~ 3600.

3 Numerical Integration

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 approximating the function by the line connecting its values on the ends of the sub-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. We show that this converges to I, assuming f is continuous:

Proof. Formally, the claim is that the sequence (T_n), where T_n is the trapezoid sum with n even subintervals, converges to I. That is, we wish to show that for all $\epsilon > 0$, there exists $N \in \mathbf{N}$ such that $|T_n - I| < \epsilon$, for all $n \geq N$. Since f is integrable with integral I, by Darboux's criterion, there exists $\delta > 0$ such that if P is a partition of $[a, b]$ with $||P|| < \delta$, then $|R - I| < \epsilon$ for all R, where R is a Riemann sum relative to P. We claim $N = \lceil (b - a)/\delta \rceil$ suffices.

Let $n \in \mathbf{N}$ be greater than or equal to N and consider the partition

$$P_n = \bigcup_{k=0}^{n+1} a + .$$

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

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

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 Line Integrals

As an application of the work in the previous section, I wrote a program (lineintegrals.py) that uses a numerical integration program (compatible with any of the ones I wrote and discussed) to compute line integrals. The vector operations are implemented by representing vectors as lists in Python. For example, a vector field in $\mathbf{R}^3$ is built by writing a function that takes in a list (representing coordinates) and returns a list (representing the vector value of the field at that point). Paths are represented parametrically and the velocity vector is found using the finite difference techniques discussed previously. The program then creates a function approximating the integrand (taking the dot product) and passes it to a numerical integration program to compute the line integral.

5 Sources

1. *Computational Science and Engineering*. Strang, Gilbert. 2007.

2. *Numerical integration and the redemption of the trapezoidal rule*. Johnson, S. G. 2011. MIT Applied Math, IAP Math Lecture Series 2011. https://ocw.mit.edu/courses/18-335j-introduction-to-numerical-methods-spring-2019/resources/mit18_335js19_lec33_1/

3. *Gradient descent, how neural networks learn*. Sanderson, Grant (3Blue1Brown). 2017. https://www.youtube.com/watch?v=IHZwWFHWa-w

The centered difference trick in section 1 I learned from Strang. The trapezoidal rule I remembered vaguely from my calculus course and derived it from there. The proof of its convergence I generated using techniques I learned auditing real analysis. I found out about Clenshaw-Curtis quadrature in Johnson after searching for improvements on the trapezoidal rule. The theoretical convergence rate of the trapezoidal rule is from there. I watched Sanderson's video on gradient descent about a year ago, but didn't refer to it while implementing it.