CS 3430: S24: Scientific Computing
Lecture 08
Differentiation and Differentiation Engines: A Dive into Symbolic Mathematics: Part 01

Vladimir Kulyukin
Department of Computer Science
Utah State University

Midterm 01 Announcement

I will post Midterm 01 in Canvas around 7:00pm on 02/07/2024. This take-home exam will be due in Canvas on 02/12/2024 by 11:59pm. This due date will allow you to complete HW04 before you work on the midterm.

The lecture on 02/07/2024 will be a review. I will not introduce any new material.

Introduction

Derivative is a fundamental tool of scientific computing, because it enables us to estimate the rate of change of a function as well as the direction of change: positive, negative, flat.

If we have a non-vertical line $f(x) = mx + b$, the $(0, b)$ is the y-intercept and m is the line's slope; for each value of x, $f'(x)$ gives the slope of $y = f(x)$ at $(x, f(x))$; the process of computing $f'(x)$ is called **differentiation**.

If the derivative of a function is positive at a point or a sequence of points, then we can say that the function is ascending on those points. If the derivative of a function is negative, then the function is descending. If the derivative is zero, then the function is flat, i.e., there is no change.

Marginal Cost: Another Interpretation of Slope

Let's consider this problem.

> `A manufacturer finds that the total cost of producing x units of a commodity is 2x + 1000 USD. What's the significance of the y-intercept and the slope of the line y = 2x + 1000?`

The y-intercept is (0, 1000); y-intercept represents the *fixed* cost for the manufacturer; this is the cost of doing business (rent, insurance, legal fees, etc.); the fixed cost must be paid even if there is no manufacturing.

Let us plug a few possible values for x: $f(1500) = 4000$; $f(1501) = 4002$; $f(1502) = 4004$; we notice that as x increases by 1, $f(x)$ increases by 2 (i.e., the actual value of the slope in $y = 2x + 100$); so the slope is the additional cost incurred when the production level is increased by 1 unit from x to $x + 1$; in economics, the term for the slope is *marginal cost*.

Review: Useful Facts About Lines

Lines can be represented in point-intercept form (i.e., $y = mx + b$) or in point-slope form (i.e., $y - y_1 = m(x - x_1)$).

Parallel lines have the same slope.

When two lines are perpendicular (excluding vertical lines), the product of their slopes is –1.

Slope as Rate of Change

Given a linear function $f(x) = mx + b$, we observe that as x varies from x_1 to x_2, $f(x)$ varies from $y_1 = f(x_1)$ to $y_2 = f(x_2)$.

The rate of change of $f(x)$ over the interval from x_1 to x_2 (i.e., $[x_1, x_2]$) is (change in y)/(change in x). In other words,

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{Dy}{Dx} = m,$$

where Dy is the vertical change and Dx is the horizontal change.

Anatomy of Differentiation Engine

The systems that compute derivates are called *differentiation engines*.

A typical differentiation engine consists of the following three main modules:

1. Parser;
2. Derivative Rules Applier (DRA);
3. Lambdification Engine (LE).

The Parser takes in raw texts and produces data structures that are fed to the DRA. The DRA takes these data structures and modifies them by applying differentiation rules to them. The the LE takes the modified data structures from the DRA and produces callable functions. In short, raw text – in, a callable function – out.

Anatomy of Differention Engine

Let us make the statemement on the previous slide more concrete.

The Parser takes a text (e.g., `'5x^2 + 20'`) and outputs a data structure (i.e., an abstraction of the actual text).

The DRA takes this data structure and applies differentiation rules to it (e.g., if $f(x) = x^r$, then $f'(x) = rx^{r-1}$); there can be dozens and even hundreds of such rules in the DRA's rule base; the DRA returns another function representation and gives it to the Lambdification Engine.

The LE takes the modified data structure from the DRA and converts to a callable function. A callable function is a function object that can be called by other programs written in the same programming language such as Python.

Anatomy of Differention Engine

Since we confine ourselves to polynomials, the DRA will use three rules:

1. If $f(x) = c, c \in \mathbb{R}$, then $f'(x) = 0$.
2. If $f(x) = cx, c \in \mathbb{R}$, then $f'(x) = c$.
3. If $f(x) = x^r, r \in \mathbb{R}$, then $f'(x) = rx^{r-1}$.

More on this in the subsequent lectures.

Lambdification

Lambdification is a process of representing a computation as a function. The name originates from Dr. Alonzo Church's λ-calculus, a function representation formalism he developed in the 1930's.

The function $f(x, y) = x + y$ is represeted in λ-calculus as $\lambda xy.x + y$. In mathematics, to compute the value of the function by substitution, i.e., $f(2, 3) = 2 + 3$. We do the same in λ-calculus, but the syntax is slightly different: $(\lambda xy.x + y)(2, 3) = 2 + 3 = 5$.

Let us work out a slightly more complex example. $((\lambda x.\lambda y.x + y)(1))(2)$ $= (\lambda y.1 + y)(2) = 1 + 2 = 3$.

Lambdification in Python

Below are a few lambdification examples in Python.

```
>>> (lambda x,y: x+y)(1,2)
3
>>> (lambda x,y: x+y)(1,2) == 3
True
>>> (lambda x: x+2)(1) == 3
True
>>> (lambda y: 2+y)(1) == 3
True
>>> ((lambda x: lambda y: x+y)(1))(2) == 3
True
>>> ((lambda x: lambda y: x+y)(1))(2) == 3
True
>>> f = (lambda x: lambda y: x+y)(1)
>>> f(1)
2
>>> f(2)
3
>>> f(100)
101
```

Polynomials

We will confine our function representation to polynomials with real coefficients. For example, $5x^2 + 10x - 10$ and $120x^{10} + 5x^7 - 3x^2 + 15x - 1000$.

In general, if $a_0, \cdots, a_n$ are real numbers, then a polynomial can be defined as follows.

$$a_0x^0 + a_1x^1 + \cdots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1} + a_nx^n = \sum_{i=0}^{n} a_ix^i.$$

Representation of Polynomials as Data Structures

To represent polynomials, we need to represent (i.e., implement in Python) the following concepts:

1. **constant** – this will be saved in **const.py**;
2. **variable** – this will be saved in **var.py**;
3. **power** – this will be saved in **pwr.py**;
4. **sum** – this will saved in **plus.py**;
5. **product** – this will be saved in **prod.py**.

The Parser will covert texts into these data structures.

Constant (Module: const.py)

A constant is an object with a single member slot (i.e., `__val__`) that contains the actual value of the constant.

```
class const(object):
    def __init__(self, val=0.0):
        self.__val__ = val

    def get_val(self):
        return self.__val__

    def __str__(self):
        return str(self.__val__)
```

Static Methods to Add and Multiply Constants

Let us define static methods that add and multiply constants to produce other constants.

```
class const(object):
    @staticmethod
    def add(c1, c2):
        assert isinstance(c1, const)
        assert isinstance(c2, const)
        v1, v2 = c1.get_val(), c2.get_val()
        return const(val=(v1 + v2))

    @staticmethod
    def mult(c1, c2):
        assert isinstance(c1, const)
        assert isinstance(c2, const)
        v1, v2 = c1.get_val(), c2.get_val()
        return const(val=(v1 * v2))
```

Variable (Module: var.py)

A variable is an object with a single member slot (i.e., `__name__`) that contains the name of the variable.

```
class var(object):
    def __init__(self, name='var'):
        self.__name__ = name

    def get_name(self):
        return self.__name__

    def __str__(self):
        return self.__name__
```

Power (Module: pwr.py)

A power is an object with two slots that contain the base (__base__), which is a variable, and the degree (__deg__), which is a real constant, to which the base is raised.

```
class pwr(object):
    def __init__(self, base=None, deg=None):
        self.__base__ = base
        self.__deg__  = deg

    def get_base(self):
        return self.__base__

    def get_deg(self):
        return self.__deg__

    def __str__(self):
        return '(' + str(self.__base__) + \
               '^' + str(self.__deg__) + ')'
```

Sum (Module: pwr.py

A sum is an object with two slots that contain the first (__elt1__) and second (__elt2__) elements of the sum. We cannot call it sum, because it is a keyword in Python.

```
class plus(object):
    def __init__(self, elt1=None, elt2=None):
        self.__elt1__ = elt1
        self.__elt2__ = elt2

    def get_elt1(self):
        return self.__elt1__

    def get_elt2(self):
        return self.__elt2__

    def __str__(self):
        return '(' + str(self.__elt1__) + '+' \
               + str(self.__elt2__) + ')'
```

Product (Module: prod.py)

A product is an object with two slots that contain the first (__mult1__) and second (__mult2__) elements of the product.

```
class prod(object):
    def __init__(self, mult1=None, mult2=None):
        self.__mult1__ = mult1
        self.__mult2__ = mult2

    def get_mult1(self):
        return self.__mult1__

    def get_mult2(self):
        return self.__mult2__

    def __str__(self):
        return '(' + str(self.__mult1__) + \
               '*' + str(self.__mult2__) + ')'
```

Maker Functions

Let us define a set of maker functions to make object construction easier.

```
def make_var(var_name):
    return var(name=var_name)

def make_const(val):
    return const(val=val)

def make_pwr(var_name, d):
    return pwr(base=make_var(var_name), deg=make_const(d))

def make_prod(mult_expr1, mult_expr2):
    return prod(mult1=mult_expr1, mult2=mult_expr2)

def make_plus(elt_expr1, elt_expr2):
    return plus(elt1=elt_expr1, elt2=elt_expr2)
```

To Be Continued...

References

1. L. Goldstein, D. Lay, D. Schneider, N. Asmar. *Calculus and its Applications*.
2. `www.python.org`.