Lab 3:
Derivative Works

Due February 16 by the start of lecture. See **Deliverables** below.

(Adapted from *Structure and Interpretation of Computer Programs* by Abelson and Sussman.)

Overview

In this lab you will complete two separate tasks:

1. Estimate the value of π using the Monte Carlo method.
2. Extend the `derivative` function from lecture to work with additional derivative forms.

Program 1

The **Monte Carlo method** for estimating the value of π uses a random number generator to simulate the throwing of darts onto a dart board. In this program, you will simulate a large number of dart throws, and through geometric probability, estimate the value of π based on your empirical results.

Suppose we draw a circle of radius 1 with center at the origin $(0, 0)$. One quarter of that circle lies in the "first quadrant" of the Cartesian coordinate plane, where x and y coordinates are both non-negative. The area of our circle is $\pi r^2 = \pi \cdot (1)^2 = \pi$, and so the area of the quarter of the circle in the first quadrant is $\frac{\pi}{4}$. We will "throw" darts at this quarter-circle by randomly generating a coordinate pair for each dart in the interval $[0, 1]$; the area where darts might land forms a square of side length 1 and total area 1. The probability that a dart will land inside the quarter circle is found by dividing the area of the target area ($\frac{\pi}{4}$) by the area of the possible landing area (1). If we "throw" a large number of darts at our board and count the number that land inside the quarter-circle, we can estimate the value of $\frac{\pi}{4}$ and thus also π. (Assuming our random number generator is truly random, of course.)

You will create a "Point" type in Clojure using the constructor/selector/operation pattern from lecture (the Vector2D examples). You will then program a Monte Carlo simulation in Clojure by implementing the following functions:

1. `make-point` - a constructor for a point in the 2D plane, with x and y coordinates.
2. `x-coord` - a selector for a Point object's x coordinate.
3. `y-coord` - a selector for a Point object's y coordinate.
4. `random-point` - this function takes no arguments and returns a randomly generated Point with coordinates in the interval of $[0, 1]$. See `rand`.
5. `throw-darts` - takes a single argument `n` representing how many darts to throw, then generates a list of Points of length `n`. Use `repeatedly` to generate an infinite sequence of `random-point` calls; use `take` on the result to collect only the first `n` results.
6. `is-hit?` - takes a Point representing a dart, and returns whether that dart falls inside our quarter-circle. Hint: calculate the distance from the origin to the dart's coordinates, and decide if that distance means the dart lands inside or outside the quarter-circle. Use selectors for the Point's attributes, and **not** `first` / `next` directly.
7. `count-hits` - takes a single argument `n` representing how many darts to throw, "throws" that many darts, and `counts` how many of those throws landed in the quarter-circle.
8. `estimate-pi` - takes a single parameter `n`; uses `count-hits` and correct mathematics to estimate the value of π.

Program 2

Starting with the code given to you in the Clojure repository, extend the `derivative` example by following these instructions:

1. Implement the Difference Rule for functions: $\frac{d}{dx}\left(f\left(x\right)-g\left(x\right)\right)=f'\left(x\right)-g'\left(x\right)$.
   (a) Add these methods:
      i. `(diff? form)`: returns true if the form is a difference.
      ii. `(make-diff a b)` returns a list representing $a-b$.
      iii. You can use `addend` and `augend` as if the form were a sum.
   (b) Expand the `derivative` function to test a new condition to see if a given form is a difference. If it is, construct the result of applying the Difference Rule using `make-diff`, `addend`, `augend`, and `derivative`.
2. Implement the Power Rule for polynomial terms: $\frac{d}{dx}\left(u^n\right)=n\cdot u^{n-1}\cdot\frac{d}{dx}\left(u\right)$, where n is a real number not equal to 0.
   (a) We will use the `**` symbol for powers, as in `(** x 2)` being x^2, and `(** (+ (* 5 (* x x)) 1) 5)` being $\left(5x^2+1\right)^5$.
   (b) Add these methods:
      i. `(power? form)`: returns true if the form is a power function.
      ii. `(make-power a b)`: returns a list representing the power function a^b.
      iii. `(base power)`: returns the base of the power function a^b (which is a).
      iv. `(exponent power)`: returns the exponent of the power function a^b (which is b)
   (c) Expand the `derivative` function to test a new condition to see if a given form is a power function. If it is, construct the result of applying the Power Rule using `make-prod`, `make-power`, and `derivative`.
3. Implement the Quotient Rule for derivatives: $\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right)=\frac{g(x)\cdot f'(x)-f(x)\cdot g'(x)}{\left(g(x)\right)^2}$
   (a) Add these methods:
      i. `(quot? form)`: returns true if the form is a quotient.
      ii. `(make-quot a b)`: returns a list representing the quotient $\frac{a}{b}$.
      iii. `(dividend quot)`: returns the dividend (numerator) of the quotient `quot`.
      iv. `(divisor quot)`: returns the divisor (denominator) of the quotient `quot`.
   (b) Expand the `derivative` function to test a new condition to see if a given form is a quotient. If it is, construct the result of applying the Quotient Rule using `make-quot`, `make-prod`, `make-sum`, `dividend`, `divisor`, and `derivative` as appropriate.
4. Implement the logic for derivatives of the natural logarithm `ln`: $\frac{d}{dx}\left(\ln\left(u\right)\right)=\frac{\frac{d}{dx}(u)}{u}$
   (a) A logarithm form will be represented as `(ln ___)`.
   (b) Add `methods` for `ln?` and `log-of` (returns the u of $\ln\left(u\right)$).
   (c) Apply the rule in `derivative` as before, using `make-quot`, `log-of`, and `derivative`.
5. Finally, we note that our output often isn't very pretty:

```
(derivative '(+ (* x x) 5) 'x)
=> (+ (+ (* x 1) (* 1 x)) 0)
```

   which says that the derivative of x^2+5 is $x\cdot 1+x\cdot 1+0$. This is correct, but certainly not preferred. Intelligent simplifying of math expressions is a very difficult problem, but there are a few simple rules we can apply to make this output nicer:

(a) Modify `make-sum` so that if either `a` or `b` is a number equal to 0, then the other value is returned, **instead of** a list representing a sum function. That is, `(make-sum 'x 0)` should return `x`, and not `(+ x 0)`. If both `a` and `b` are numbers, then return the sum of those numbers **instead of a list**.

(b) Modify `make-diff` so that if `a` is 0, then then list `(- b)` is returned, and of `b` is 0, then `a` is returned. If both `a` and `b` are numbers, return their difference.

(c) Modify `make-prod` similarly, so that a product with 0 returns 0, and a product with 1 returns the other value. If both `a` and `b` are numbers, return their product.

(d) Modify `make-quot` similarly, so that a divisor of 1 returns the dividend, and a dividend of 0 returns 0. If both `a` and `b` are numbers, return their quotient.

(e) Modify `make-power` similarly, so that a base raised to the power of 1 returns the base, and a base raised to the power of 0 returns 1. If both `a` and `b` are numbers, return `(Math/pow a b)`.

424H Requirements

If you are enrolled in CECS 424H, then you must complete the following requirements in addition to the requirements above.

1. Implement the logic for the derivatives of the simple trigonometric functions:
   $\frac{d}{dx}(\sin(u)) = \cos(u) \cdot \frac{d}{dx}(u)$,
   $\frac{d}{dx}(\cos(u)) = -\sin(u) \cdot \frac{d}{dx}(u)$,
   $\frac{d}{dx}(\tan(u)) = \sec^2(u) \cdot \frac{d}{dx}(u)$

   (a) A trigonometric form will be represented as `(sin ____)`,`(cos ____)`, or `(tan ____)`.

   (b) Add methods for `sin?`, `cos?`, `tan?`, `make-sin`, `make-cos`, `make-sec`, and `angle` (returns the u of any trig form).

   (c) Apply the rules in `derivative`, using `make-prod`, `make-diff`, `make-sin`, `make-cos`, `make-sec`, and `derivative`.

2. Add derivative rules for the exponential function: $\frac{d}{dx}(b^u) = b^u \cdot \ln(b) \cdot \frac{d}{dx}(u)$

3. Modify the differentiation program so that it works with ordinary mathematical notation, in which + and * are infix rather than prefix operators. Since the differentiation program is defined in terms of abstract data, we can modify it to work with different representations of expressions solely by changing the predicates, selectors, and constructors that define the representation of the algebraic expressions on which the differentiator is to operate. To simplify the task, assume that + and * always take two arguments and that expressions are fully parenthesized.

Deliverables

Turn in the following when the lab is due:

1. A printed copy of your Monte Carlo code, **printed from your IDE when possible.** If you cannot print from your editor, copy your code into Notepad or another program with a fixed-width (monospace) font and print from there.

2. A printed copy of your derivatives code, **printed from your IDE when possible.** If you cannot print from your editor, copy your code into Notepad or another program with a fixed-width (monospace) font and print from there.

3. Run your `estimate-pi` function several times with these values for `n`. Turn in the function's output for each:

   (a) 10

   (b) 100

(c) 1000

(d) 100000

Then, **explain why your estimate(s) at small values of `n` are so poor.**

4. Run your `derivative` function with the following inputs, and turn in the output. For each function call, give the exact output from your program, then **translate** that output into formal mathematics notation, e.g., `(+ (** x 2) 5)` would be translated to $x^2 + 5$.

   (a) `(derivative '(- (* 5 x) y) 'x)`

   (b) `(derivative '(** (ln (+ (** x 2) 1)) 5) 'x)`

   (c) `(derivative '(/ (* 5 (sin x)) (cos x)) 'x)`