Project 0: Prerequisites

Deliverables

For this assignment you need to complete the Julia programs `matrices.jl` and `gradients.jl`. Your GitHub repository will become your submission, so commit your work as you go.

Partners and Groups

This is an individual assignment. You are encouraged to talk with your classmates about Julia syntax and other general concepts, but you should be writing your own code. Next week, you'll do a code review with a randomly-assigned classmate, so make sure to spend some time commenting your code and making it readable.

Objectives

This project has three primary goals; it aims to help you:

1. get familiar with programming in Julia,
2. review some concepts from multivariable calculus, and
3. review some concepts from linear algebra.

Unfortunately, these objectives are slightly in conflict. Much of what makes Julia awesome is the ability to succinctly express complex mathematical concepts, but in order to help you review the prerequisite concepts, you are being asked to implement them from scratch rather than using libraries that would do all the work for you. Please pay close attention to the requirements expressed below: there are lots of shortcuts that exist for these problems that you are being deliberately instructed to avoid. Take a look at the unit tests (especially `test_matrices.jl`) and the library documentation (especially `TypedPolynomials`) for better examples of how to do these things in practice.

Gradients

In the file `gradients.jl` you have three functions to implement:

- `evaluate()` finds the value of a multivariate polynomial at a given point
- `partial_derivative()` finds the derivative with respect to a given variable
- `symbolic_gradient()` finds the vector of partial derivatives for each variable

Each of these functions takes an object of type `Polynomial` from the library `TypedPolynomials`. You will need to add this library, and then you should try out the following code one line at a time in a Jupyter notebook:

```
@polyvar x y z
p = 2x + 3.0x^2*y^2 - y*z^4
p(x=>3, y=>-4, z=>.05)
differentiate(p, x)
```

As this demonstrates, the library already implements ways of evaluating and differentiating multivariate polynomials, and you are encouraged to experiment with this, and use it to test against, but you will be writing your own version of this functionality.

You are not allowed to use the differentiate function, nor are you allowed to call `p()` to evaluate a polynomial `p`. You are also not allowed to use the $\div$ operator on polynomials; instead you must construct polynomials using only the $+$, $*$, and $\wedge$ operators.

To do this you will need to access the internals of `Polynomial` objects. The following operations demonstrate most of what you will need:

```
variables(p)
p.terms
p.terms[1].coefficient
p.terms[1].monomial
p.terms[1].monomial.exponents
```

Try these out in a Jupyter notebook to get a sense of what they do. You are welcome to implement any helper functions you think would be useful. Your eventual goal is to pass all of the tests in `test_gradients.jl`, but you should be testing on your own as you go. Specifically, I recommend working on one function at a time, in a scratch-work Jupyter notebook where you've also declared some example polynomials to test on. When you get a function mostly working, you can transfer it over to `gradients.jl` and then use `include` to load it.

Matrices

In the file `matrices.jl` you have four functions to implement:

- `dot_product` computes the dot product of two vectors
- the first version of `multiply` computes the product of a matrix and a vector
- the second version fo `multiply` computes the product of two matrices
- `p_norm` computes a vector norm

These functions all exist as built-in operations in Julia's `Base` or `LinearAlgebra` packages. However, you are not allowed to use those implementations. Instead, you should implement each of these functions down to single-element arithmetic operations. Each function should use a single for loop. Start with `dot_product`, then use this function when implementing matrix-vector products and use your matrix-vector product function when implementing matrix-matrix products.

For the first three functions, it is possible to have invalid inputs. Try out the following in Jupyter:

```
a = ones(4,2)
b = collect(1:4)
a * b
```

Your implementation should mimic the standard one by throwing an error in such cases. You are welcome to research the types of errors the built-in operations can throw and reproduce them, but that is not required. The unit tests only verify that an exception occurs, so using `@assert` is the recommended approach. Once again, you are strongly encouraged to use a Jupyter notebook for incremental testing and development as well as scratch work!