An introduction to Rust

Foreword

This session is somewhat a bias introduction from me because I love `Rust`. Here is the backstory: I was first introduced to `Rust` in 2022, mostly because of the controversy around it, and I did part of the advent-of-code (AOC) in Rust. At the time, I could not see much point of learning and using `Rust`, mostly because I do not know where to use `Rust`. I went on a year without using and following `Rust`, then I did most of AOC 2023 in `julia`. On December 24, 2023, I came across an article about web assembly (`wasm`), which talks about how `wasm` can be used to distribute code on the web which run on client-side with almost native performance, and how one of the main use case for `Rust` is to compile to `wasm`. I unfortunately and willingly sacrifice my 25th day of AOC to learn about `Rust` and `wasm`, and it was awesome. For the next couple of months I was exploring `Rust` a lot, and it was such a fun experience.

In case you do not realize how interesting it is, let me break it down for you. If you have supervised students or collaborate with others on a code project before, especially with someone using a MacOS, the most difficult part of the job could be getting them to install the dependencies correctly. Nowaday most python package can be install with `pip install [package]`, but sometimes people need to build a package dependencies from source, and that is a highway to nightmare if the code is not supported by a huge community. Being able to write in a programming language like `Rust`, compile it to `wasm` and run it directly on a browser without any installation is a complete game changer. No install needed, just open the browser and boom, the code runs. Say in a dire situation like needing to factor some prime numbers in a party, you can just pull out your phone, open the browser and check whether a number is a prime. Isn't that cool?

Beside, I really had a great time coding Rust in general. It takes some time to get used to the language, but once you are more familiar with Rust, it is a great language to go hard core with. In this lab, we are going to go through the following topics:

Key Concepts

Rust is a compiled language

Unlike `python` and `julia` we have been playing with in the last two sessions, `Rust` does not offer a REPL, and it is not designed to be an interactive language. Instead, `Rust` is an ahead-of-time compiled language, which means you need to compile the code before you can run it. Similar to `c` and `c++`, `Rust` code is compiled into machine code and into executable, which then the users can execute.

Rust is strongly and statically typed

Another major difference between `Rust` and the other two languages is that `Rust` is a strongly and statically typed language. This means that the type of every variable must be known at compile time, and the compiler will check that the types are used correctly. If you try to assign a variable of type `i32` to a variable of type `f64`, the compiler will throw an error. However, it does offer type inference, meaning the compiler will try to figure out the type of the variable if it is not explicitly stated. This is not gaurenteed to work all the time, but it is a nice feature to have.

Borrow checker for memory management

An introduction to Rust

The biggest innovation Rust brings to the table of programming language is the concept of ownership, or more commonly referred as the borrow checker[1].

Basic Syntax

Variables

Scalar types

As opposed to `python` and `julia`, `Rust` requires you to declare the type of a variable when you define it[2]. Here is an example of how you would declare a variable in `Rust`:

```
let x: i32 = 5;
```

In this example, we declare a variable `x` of type `i32` and assign it the value `5`. The `let` keyword is used to declare a variable, and the `:` is used to specify the type of the variable. The `i32` type is a 32-bit signed integer. You may be tempted to leave out the type declaration and let the compiler infer the type, but here is an example that shows why it is a good idea to specify the type explicitly:

```
let guess = "42".parse().expect("Not a number");
```

In this example, we try to convert a string into numeric type, but without specifying what is the type of the variable `guess`, the compiler will not know what type to convert the string to. This will result in a compilation error.

Another thing to pay attention to is mutability. By default, variables in `Rust` are immutable, meaning you cannot change the value of the variable once it is assigned. If you want to make a variable mutable, you need to use the `mut` keyword:

```
let mut x = 5;
x = 6;
```

Compound types

Compound types such as tuple and array are also available in `Rust`, and they are quite different from the other languages we have seen so far. Here is an example of how you would declare a tuple in `Rust`:

```
let tup: (i32, f64, u8) = (500, 6.4, "a");
println!("The value of y is: {}", tup.1);
```

You have to declare the type of each element in the tuple, and the type of the tuple itself is a combination of the types of its elements. We can access the elements of the tuple using the `.` operator followed by the index of the element we want to access.

Moving on to arrays. The definition of an array in `Rust` is different again. In `python` and `julia`, you can create an array without specifying its length at initialization. However, the compiler will need to know about the length of the array at compile time:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

And the elements of the array can be accessed using the `[]` operator:

[1] Most people encounter the concept of ownership through the borrow checker.
[2] While type inference does exist in `Rust`, it is still a good practice to write the type of your variable explicitly.

```
let first = a[0];
```

Note that all the types we have discussed so far are fixed-length, and they are allocated to the stack, which is faster than the heap but do not allow for dynamic resizing.

To create

If you want to create an "array" with variable length, you can use the `Vec` type:

Functions

Control flow

Scope and borrowing

We will write a simple MCMC algorithm this time

To switch things up a bit, instead of coding an insertion sort yet again, we will code a simple Metropolis-Hastings algorithm in Rust. More specifically, we are going to sample from a Gaussian distribution with a Gaussian proposal distribution.

Step 0: Install Rust

Follow the instruction on this link. After installing Rust, you should have access to the `cargo` command in the terminal. If you are using VSCode as your IDE, you can also install the `Rust-analyzer` extension to get better support for Rust.

Step 1: Clone the class repository

Clone the template repo from this link.

Step 2: Implement the Metropolis-Hastings algorithm

The pseudocode for the Metropolis-Hastings algorithm is as follows:

1. Start at an arbitrary point `x`
2. Repeat the following steps for `n` iterations:
   1. Sample a candidate point y from a Gaussian distribution with mean x and variance σ, $y \sim N(x, \sigma^2)$
   2. Calculate the acceptance probability $p = \min\left(1, \frac{p(y)}{p(x)}\right)$, where p is the target distribution
   3. Sample a uniform random number $u \sim U(0, 1)$
   4. If `u < acceptance probability`, set `x = y`
   5. Store `x` in a list

There should be some starter code in the `sampler.rs` file, which contains the following functions: a struct named `State` to store the current state of the sampler, a function named `long_likelihood` as our target function, implementation of the `State` struct with a `new` function, and a `take _step` function to sample a new state from the proposal distribution. We will implement them one by one in the following section.

Step 2.1: Creating a struct for the sampler state

Step 2.2: Implementing the target function

Step 2.3: Implementing the `new` function in the `State` struct

Step 2.4: Implementing the `take_step` function in the `State` struct

Step 3: Test the algorithm

While `Rust` is an awesome language and give developer a lot of low level access such as GPU programming, interactive workflow such as making a plot in a data science workflow is not `Rust`'s main focus. There are some visualization library in `Rust` such as `plotters`, but in order to visualize the result we have generated, the script needed is linked here, which is quite long compared to python. For people who want to try it out, go for it. But this part, we are just going to run a scatter plot with `python` and `matplotlib`.

Once you have implemented algorithm as stated in Step 2, run the following command in the terminal to generate the data:

```
cargo run | test.text
```

This should run your main function and dump the output to a file called `test.text`. Now, run the following python script to visualize the result:

```
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt("test.text")
plt.scatter(data[:, 0], data[:, 1])
plt.show()
```

Now you saw the message saying the code is compiled but not optimized, and you are wondering why. The reason is there are different optimization configurations that offers different trade-offs between compile time and runtime performance. To compile the code with optimization, run the following command:

```
cargo run --release | test.text
```

This code should run faster than the previous one.

Putting the MCMC algorithm on the Web

Now you have familiarized yourself with the basic syntax of `Rust`, let's get to the **really** fun part: serving your code as a client-side web application. We are going to compile our `Rust` code into `WebAssembly` (`wasm`) and run it in the browser. While `WebAssembly` itself is a programming language, it is more maded to be a compilation target for other languages such as `Rust`, `c`, `c++`, meaning you can write in `Rust` then compile the code into `wasm`. WebAssembly provides a way to run code in the browser at near-native speed on client side.

Step 0: Installing wasm-pack

The package we will use to bundle our `wasm` code such that we can use in a normal web development workflow is called `wasm-pack`. To install `wasm-pack`, run the following command:

```
cargo install wasm-pack
```

Step 1: Restructuring the code

Step 2: Scaffolding the frontend

Since we are going to learn more about frontend development in the future session, we are not going to get fancy here. Instead, we are going to create the bare minimum needed to demonstrate we can run our `Rust` code in the browser.

Step 3: Calling our binary

Best practices

Development tips

Use linter/formatter/extensions

Noteworthy libraries

Bevy

One thing you can do

Axum/Actix

wGPU

candle/burn

Yew/leptos/dioxus

There are a number of frontend frameworks in `Rust` that are under active development. The biggest frontend framework so far is Yew, which is a frontend framework inspired by `React`. The next two that are pretty similar in community size are "leptos" and dioxus. While they all sounds pretty cool and can be fun to play with, none of them have reached version 1.0 yet. This means their API is not yet stable and could cause some hassle in your development. For this reason, I am staying with javascript for frontend development, and only use `Rust` whenever I have performance critical functions.