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.

An introduction to Rust

Borrow checker for memory management

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]. This is to address the age old problem of memory management. There are usually two routes different programming languages take to manage memory: garbage collection and manual memory management. Both `python` and `julia` have a garbage collector that will regularly look for unused memory and release them. This is a very convenient way to manage memory, but it comes with overheads. On the other hand, languages like `c` demands the user to manually manage the memory allocation and deallocation, which could create segmatation fault if not done correctly. `Rust` takes a different approach: it uses the concept of ownership to manage memory. The idea is that every value in `Rust` has a variable that is its owner, and there can only be one owner at a time. When the owner goes out of scope, the value is dropped, and the memory is deallocated. We will explore this more in the next section.

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);
```

[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.

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:

```
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.

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

```
let v: Vec<i32> = Vec::new();
for i in 1..5 {
    v.push(i);
}
// You can also do this: let v = vec![1, 2, 3, 4, 5];
```

The `Vec` type is a growable array type that is allocated on the heap. This means that the size of the array can change at runtime, and it is slower than the fixed-length arrays. There are similar data type like `string` and `hashmap` that are also allocated on the heap. See here for more information.

Functions

To write a function in `Rust`, you need to use the `fn` keyword followed by the name of the function and the arguments it takes. Unlike `python`, you need to specify the type of the arguments and the return type of the function, otherwise the compiler will throw an error. Here is an example:

```
fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

Note that the last expression in the function is the return value of the function, and you do not need to use the `return` keyword. You can still use the `return` keyword if you want to return early from the function.

Control flow

Compared to all the definitions you have seen, the syntax of control flows in `Rust` is actually pretty similar to other languages. For example, to write a `for` loop in `Rust`, you can do the following:

```
for i in 1..5 {
    println!("{}", i);
}
```

This is very similar to the `for` loop in `python` and `julia`. The `1..5` syntax is a range, which can be iterated through. Since this is pretty simple and straight forward, we are not going to spend more time on this topic. For more detail on control flow, see here.

An introduction to Rust

Scope and borrowing

Now let's get to the fun part: ownership. This is `Rust` unique way to manage memory, and it is the biggest reason why others consider `Rust` a very safe language. If you come from a `c` background and you maybe somewhat familiar with this syntax, but there should be a significant difference in terms of how `Rust` handle memory compared to `c`.

The basic problem statement of memory management is basically you need to allocate a chunk of address to store some of the values related to your computing task, and once you are done with the task, you should release the memory back to the system. In `c`, it could be done by

```c
int *x = (int *)malloc(sizeof(int));
*x = 5;
free(x);
```

Here, we allocate the memory space for an integer, assign the value `5` to it, and then free the memory space. However, no matter how experience you are, you may eventually run into some memory issue. For example, if you try to access a memory that has been deallocated, you will get a segmentation fault. For example,

```c
int *x = (int *)malloc(sizeof(int));
*x = 5;
free(x);
printf("%d\n", *x);
```

This code will compile, but when you run it, you will get a segmentation fault. This is because the memory space that `x` points to has been deallocated, and you are trying to access it.

In `Rust`, there is a very smart approach to handle this. The idea is that every value in `Rust` has a variable that is its owner, and there can only be one owner at a time. When the owner goes out of scope, the value is dropped, and the memory is deallocated. Here is an example:

```rust
{
    let x = 5;
    println!("{}", x);
}
```

In this example, the curly braces define a new scope, and the variable `x` is the owner of the value `5`. When the scope ends, which is when the closing curly brace is reached, the value `5` is dropped, and the memory is deallocated. And if you try to access the value of `x` after the scope ends, you will get a compilation error, so you will find out your mistake way before you hit it in runtime. In this way, the users do not need to worry about memory managment (as much), and we don't have to pay the price of garbage collection.

Now this comes with some counterintuitive behavior for beginners. Let's see how we will 'borrow' a variable in `Rust`:

```rust
fn modify(x: &mut i32) {
    *x = 5;
}

fn main() {
    let mut x = 0;
    modify(&mut x);
```

```
    println!("{}", x);
}
```

You see the `&` symbol in front of the variable `x` in the `modify` function. This is called a reference, and it is a way to borrow a variable in `Rust`. The `&mut` keyword means that the reference is mutable, which means that the function can modify the value of the variable. The `*` symbol is used to dereference the reference, which means to get the value that the reference points to. In this example, the `modify` function takes a mutable reference to an integer, and it sets the value of the integer to `5`. If you try to remove the `&mut` keyword in the `modify` function, you will get a compilation error, because you are trying to modify a borrowed value, and that is not mutable.

Now there are more intrincate examples to really show the reason why this way to manage memory is out right awsome on the official documentation. I highly encourage you read through that as it will allow you to understand `Rust` deeper. And then the concurrency chapter is my favorite chapter that makes this way of handling memory really shines thorugh.

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

In the `sampler.rs` file, you will see a struct named `State` that is used to store the current state of the sampler. We need three fields in this struct: a random number generator `rng`, an array containing the current state `x`, and the proposal distribution.

I have import some relevant libraries on the top of the file, try to look through them and understand how why they are imported. By the end of this step, your `State` struct should look like this:

```
pub struct State<const N_DIM: usize> {
    rng: StdRng,
    pub arr: [f64; N_DIM],
    proposal_distribution: MultivariateNormal,
}
```

Notice the `pub` keyword in front of the struct, this is to make sure one can import the struct from other files. Another fancy syntax I put it is the `const N_DIM: usize`. This is called generics parameters in `Rust`, and it allows you to write functions or struct over a range of types in order to reduce code duplication.

Step 2.2: Implementing the target function

For the target distribution, let's just choose a simple Gaussian distribution with mean 0 and variance 1. Instead of specifying the probability function itself, it is often more practical to specify the log of the probability function. Try implementing the `log_likelihood` function that corresponds to

$$\log p(x) = -\sum_{i=0}^{n} \frac{x_i^2}{2}$$

. This should be easy enough that I don't want to give away the answer here.

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

The next thing we have to do is to implement the body functions related to the `State` struct. The first function we are going to implement is the `new` function, which is used to initialize the state of the sampler.

To initialize the random number generator, we will use the `StdRng` struct from the `rand` crate. It is always a good practice to set a seed for your random number generator, so that you can reproduce the results later. The function needed for generating such random number generator is `SeedableRng::from_seed`. Here is the doc page to this function.

The next thing we need to do is to initialize the array `arr` with zeros. This step should be trivial enough so answer is not given here.

Finally, we need to initialize the proposal distribution. As shown in the definition for the struct, the `proposal_distribution` is of the type `MultivariateNormal` as defined in `statrs`. The function signature should look like this:

```
let proposal_distribution = MultivariateNormal::new(mean, var).unwrap();
```

where `mean` and `var` are vectors of length `N_DIM` and `N_DIM x N_DIM` respectively. This function returns a `Result` type, which is commonly used for handling potential errors. To extract the actual object we need, we have to unwrap the result.

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

The last thing we have to implement is the `take_step` function, which is used to sample a new state. To sample a propsal state from the proposal distribution, use these follwoing lines:

```
let binding = self.proposal_distribution.sample(&mut self.rng);
let proposal= binding.as_slice();
```

The next thing you need is to compute the log likelihood ratio between the proposed location and the current location, then draw a random number from a uniform distribution to decide whether to accept the proposal. If you decide to accept the proposal, then you have to update the current location to the proposed location. There are some tricky bits here which I want to leave for you to figure out. But if you get stuck in this process, the completed code after this section is in the `MCMC` branch of the template repo.

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 add wasm-pack
```

Step 1: Restructuring the code

In order to compile our code into `wasm`, we need to adda couple more functions to the `lib.rs` file and modify the `Cargo.toml` file. Starting with the `Cargo.toml` file, we need to add the following lines:

```
[lib]
crate-type = ["cdylib"]
```

This compiles the code into a dynamic library that can be load at runtime, and in this way it can be used in the browser.

The `lib.rs` file should look like this:

```
pub mod sampler;

use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn run() {
    const N_STEP: usize = 100;
    const N_DIM: usize = 2;
    let mut state = sampler::State::<N_DIM>::new(0xdeadbeef);
    for _ in 0..N_STEP {
        state.take_step();
        log(&format!("{:?}, {:?}", state.arr[0], state.arr[1]));
    }
}

#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
    // Use `js_namespace` here to bind `console.log(..)` instead of just
    // `log(..)`
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen]
pub fn greet() {
    alert("Hello from wasm");
}
```

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.

We basically need two files: `index.html` and `index.js`. The `index.html` file will be defining the end point for where the `wasm` code will be loaded, and the `index.js` file will be loading the `wasm` code and running it. Here is the content of the `index.html` file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello World - Rust</title>
    <script type="module" src="./index.js"></script>
  </head>
  <body></body>
</html>
```

This file is as barebone as it gets. It is just a simple HTML file that includes the `index.js` file. The `index.js` file will be responsible for loading the `wasm` code and running it. Here is the content of the `index.js` file:

```
// Import our outputted wasm ES6 module
// Which, export default's, an initialization function
import init from "./pkg/mcmc_example.js";

const runWasm = async () => {
  // Instantiate our wasm module
  const helloWorld = await init("./pkg/mcmc_example_bg.wasm");

  // Call the run function in the wasm module, which will return the result of our
calculation
  helloWorld.run();
};
runWasm();
```

This file is also very simple: it imports the `init` function from the `mcmc_example.js` file, which is generated by `wasm-pack`, and then calls the `init` function with the path to the `wasm` file. The `init` function will return a promise that resolves to the `wasm` module, which we can then use to call the `run` function.

Step 3: Magic moment

If you start a runtime with the following command at the root of the project (where the html file is located), say with `python -m http.server`, you should be able to see the result of the MCMC algorithm in console of the browser.

Development tips

Make sure you check how active a crate is

`Rust` eco-system kind of have the same problem as `julia` from time to time, that is there could be multiple crates trying to do the same thing and you may pick a crate that sounds good on paper but may not actually solve your problem.

Use linter/formatter/extensions

There are great IDE extension for `Rust` that can help you write better code. For example, the `Rust-analyzer` extension for `VSCode` is a great tool that will save you a lot of fustrations. It highlights codes that will rise a compilation error, and it also helps you format your code, plus a bunch of other features. Highly recommended.

Don't forget `Rust` is a low level language

Coming from `python` or `julia`, you may have several habits on relying on existing packages or try to think in a vectorized way, i.e. "how should I use numpy instead of writing my own code?". This is completely valid, and more often than not encouraged, because there might be people with more experience working on the same algorithm who have made a crate already. However, in the end of the day, the only thing that matters is whether you code does what you want. If you are confident in the vision of your program and in how to implement the corresponding algorithm, doing it yourself maybe faster than looking up a solution, since `Rust`'s community has very different focus compared to other more data science focused languages.

Noteworthy libraries

These libraries I mention below are not necessary considered packages for data science directly. However, they give you a good sample of what `Rust` is capable of.

Bevy

Bevy is a "data-driven" game engine that allows you to build game fully with `Rust`. Since it is still in its early days, I don't usually consider it a fully production ready game engine like `Unreal`, `Unity` or event `Godot`. However, I do find it quite educational and light weight enough to be a platform to add interactibility to a demo. Because everthing is in `Rust`, it is actually great for building data-driven demos, since you can just build your data processing code in `Rust`, and then use `Bevy` to visualize the result.

Axum/Actix

`Axum` and `Actix` are the two most popular web framework in `Rust`. If you want to build a backend that handls compute and requests, they are both pretty solid options. From what I have read around, `Axum` seems to have more support around it, tho the first framework I picked up was `Actix` simply because it has a nonstandard standalone documentation page.

wGPU -> rerun.io

wGPU is a graphic library for `Rust` that is based on the WebGPU API. Despite this description as a "graphic library", it is more like a low level API that allows you to interact with the GPU directly[3].

[3] Similar to Vulkan if people have experience with that.

When combined with `wasm`, it allows you to ship computation through browser that run on the client's GPU. One example is this demon site, which uses your GPU to simulate a 2D n-body problem.

In preparing this lecture note, I also discovered rerun.io, which is mostly written in `Rust` and uses `wGPU` for rendering, it is pretty sick.

candle/burn

`candle` and `burn` are two machine learning libraries that I think they are pretty interesting. `candle` has more stars on GitHub, and it is under the hugging face organization. There are many existing production level examples you can look into, even pretty recent models such as Segment Anything. On the other hand, burn is more experimental and they are trying to really do everything from scratch in `Rust`. For example, they have a lot of worked done on asynchronous execution, which could make it a very interesting option for embedded systems in the long run.

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.