Rust

Programming Language

- Why Rust?
  - Advantages of static typing
  - Easy speedup
- How Rust?
  - Brief introduction
  - Accelerate Python
  - Easy multi-threading

Dynamic Typing

Dynamic language:

- Types are cast automatically
- Python:
  “if it quacks it's a duck”
- Less Keyboard action
- More Errors

```
##### Example 1 #####

def work_on_array(arr, idx_start=None, idx_end=None):
    """ Quick example of how duck typing can go wrong. """
    if idx_start and idx_end:
        return np.sum(arr[idx_start:idx_end])
    return np.sum(arr)

assert work_on_array([1, 1, 1, 1, 1], start=0, end=3) == 5

##### Example 2 #####

def inner_product(vec1, vec2):
    return np.sum(vec1 * vec2)

vec1, vec2 = np.ones(10), np.ones(10)
assert inner_product(vec1, vec2) == 10.0
# Outer product by mistake
assert inner_product(vec1, vec2.reshape(-1, 1)) == 100.0
```

Static Typing

- Types are defined for:
  - Variables
  - Function parameters
  - Function return values
- Helps with optimization
- It’s just how the CPU works

```rust
// ##### Some Numeric Types #####
let types_num: [u8, u64, i8, i64, f32, f64, usize];
// ##### Some Composite Types #####
let types_comp: [Array, Tuple, Vec, String, &str, HashMap];
// ##### Define var with type #####
let var_num: u8 = 0;
// ##### Define var with type #####
let vec: Vec<f32> = vec![1.0, 2.0, 3.0];
// ##### Type inference possible -> default: f64 #####
let vec = vec![1.0, 2.0, 3.0];

// ##### Function definition #####
fn add(a: i32, b: i32) -> i32 {
    a + b
    // same as return a + b
}
```

Why Rust?

- Modern language
- Package manager
- Type inference
- Super Fast (with 1% of C)
- Not C
- No header files
- Feels Great

Getting Started

Install in a single line:

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Package Manager “cargo”

- Resolves dependencies
- Installs programms (ie “$ cargo install lsd”)

Start new project:

```
$ cargo new myproject; cd myproject
```

Getting Started

Check for errors

```
$ cargo check
```

Tip: Get overlay for editor like “rust-analyzer” for VSCode

Run with

```
$ cargo run (-- release)
```

Add dependencies to Cargo.toml file

Pointers & Borrowing

Data can be passed:

- By copying
- By reference

By reference, memory address (pointer) gets passed

Requires lots of management:

- (De)Allocate memory
- Maintain valid data
- Maintain valid pointer, else → “Segmentaion Fault”

Pointers & Borrowing

In Rust:

- Variables are immutable by default
- Can be made mutable (mut keyword)
- Data is owned by a single variable
- When owner goes out of scope, data is deallocated
- Owner can borrow data to multiple readers (like pointer)
- Owner can borrow data mutably to single writer
- → Multiple Readers / Single Writer

Structs

Bundle data of different types

Performant Structuring of code

Not possible with python/numba:

- https://github.com/lucastepper/memtools/blob/master/memtools/gle_sim.py

Work like classes

Add Methods to a struct via implementation block

Traits

Define Common behavior for structs

Like inheritance in OOP

THE abstraction layer in Rust

Used all over the standard library:

- Copy → can be passed by value via copy
- Display/Debug → can be printed
- Add → works with “+” operator
- Send → can be shared between threads

Result & Option

Rust forces explicit handling for:

- Errors
- None-Values

Both are found in standard library

→ Implemented as Enums

Lazy Solution: call .unwrap()

→ Crash program if Error/None

Cool Features

Crate ndarray

- → Numpy-inspired multi-dim arrays
- → Stored as contiguous 1-dim arrays in memory

Crate py03

- → Build Python interfaces with little boilerplate
- → Create/interact with all native python types and np.ndarrays

Crate rayon

- → Build parallel iterators easily

More Material

Very good Documentation!

- Learning:
  - https://doc.rust-lang.org/book/
- Reference standard lib:
  - https://doc.rust-lang.org/std/
- Crates:
  - https://docs.rs/

Thank you for your attention!