



# Enhancing Numerical Computing with Rust: A Paradigm Shift in Numpy Function Implementation

G52



# INTRODUCTION

- The goal of this project is to optimize numerical and scientific computing.
- Currently, Numpy is widely used for numerical and scientific computing, while Rust is known for its performance, safety, and system-level programming capabilities.
- This project brings the advantages of Rust to the data science and scientific computing community while maintaining compatibility with existing Python-based workflows.

# Performance Challenges in Numpy

- The Global Interpreter Lock (GIL) is a mechanism used to synchronize access to Python objects, preventing multiple native threads from executing Python bytecodes at once.
- The GIL is a critical aspect of Python's memory management and execution model, but it has significant implications, both positive and negative.
- Without GIL, managing memory across multiple threads could lead to data corruption and crashes due to race conditions.
- Numpy functions, when implemented in Python, often suffer from the Global Interpreter Lock (GIL) and interpreted execution which leads to slower execution in computationally intensive tasks.

# The Rust Advantage

- Rust's zero-cost abstractions and low-level control enable direct memory manipulation and efficient concurrency.
- Rust's control over memory and lack of a GIL allows for more efficient utilization of hardware resources compared to Python
- Rust's compiled code eliminates interpretation overhead, leading to faster execution
- GIL in Python limits parallelism, while Rust allows for safe concurrent execution without contention.

# Implementing Numpy Functions in Rust

- Rust's ecosystem includes libraries like ndarray and ndarray-rand, which provide similar functionality for array manipulation and random number generation.

# Matrix Multiplication

## Python Implementation

```
import time
def matmul(a, b):
    a_rows = len(a)
    a_cols = len(a[0])
    b_rows = len(b)
    b_cols = len(b[0])
    if a_cols != b_rows:
        raise ValueError("Dimension Error")
    result = [[0.0 for _ in range(b_cols)] for _ in range(a_rows)]
    start_time = time.time() # Record the start time
    for i in range(a_rows):
        for j in range(b_cols):
            for k in range(a_cols):
                result[i][j] += a[i][k] * b[k][j]

    end_time = time.time() # Record the end time
    elapsed_time = end_time - start_time
    return result, elapsed_time

matrix_a = [[1.0, 2.0], [3.0, 4.0]]
matrix_b = [[5.0, 6.0], [7.0, 8.0]]
result, elapsed_time = matmul(matrix_a, matrix_b)
for row in result:
    for val in row:
        print(val, end=' ')
    print()

print("Elapsed Time:", elapsed_time, "seconds")
```

# Rust Implementation

```
use std::time::{Instant};

fn matmul(a: Vec<Vec<f64>>, b: Vec<Vec<f64>>) -> Vec<Vec<f64>> {
    let a_rows = a.len();
    let a_cols = a[0].len();
    let b_rows = b.len();
    let b_cols = b[0].len();

    if a_cols != b_rows {
        panic!("Matrix dimensions do not match for multiplication.");
    }
    let mut result = vec![vec![0.0; b_cols]; a_rows];
    for i in 0..a_rows {
        for j in 0..b_cols {
            for k in 0..a_cols {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    result
}
```

```
fn main() {
    let matrix_a: Vec<Vec<char>> = vec![vec!['a', 'b'], vec!['c', 'd']];
    let matrix_b: Vec<Vec<f64>> = vec![vec![5.0, 6.0], vec![7.0, 8.0]];
    let start = Instant::now();
    let result = matmul(matrix_a, matrix_b);
    let end = start.elapsed();
    let duration=end.as_secs() as f64+f64::from(end.subsec_nanos())/1000000000.0;
    println!("Time taken for training: {:.6} seconds", duration);
    for row in &result {
        for &val in row {
            print!("{}", val);
        }
        println();
    }
}
```

# Advantages

- Type Safety: Rust enforces type safety by declaring and maintaining strong typing.
- Performance: Rust's control over memory layout and low-level optimizations allows for efficient numerical operations.
- Memory Safety: Rust's memory safety is enforced through ownership and borrowing rules.



# Feed Forward Neural Network Algorithm

## Python Implementation

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(inputs, weights, biases):
    hidden_layer = sigmoid(np.dot(inputs, weights[0]) + biases[0])
    output_layer = sigmoid(np.dot(hidden_layer, weights[1]) + biases[1])
    return output_layer

# Example usage
input_data = np.random.rand(1000, 10)
weights = [np.random.rand(10, 20), np.random.rand(20, 1)]
biases = [np.random.rand(20), np.random.rand(1)]
result = neural_network(input_data, weights, biases)
print(result)
```

# Rust Implementation

```
extern crate ndarray;
use ndarray::prelude::*;

fn sigmoid(x: f64) -> f64 {
    1.0 / (1.0 + (-x).exp())
}

fn neural_network(inputs: Array2<f64>, weights: [Array2<f64>; 2], biases: [Array1<f64>; 2]) -> Array2<f64> {
    let hidden_layer = sigmoid(inputs.dot(&weights[0]) + &biases[0]);
    let output_layer = sigmoid(hidden_layer.dot(&weights[1]) + &biases[1]);
    output_layer
}

fn main() {
    let input_data = Array2::<f64>::random((1000, 10));
    let weights = [Array2::<f64>::random((10, 20)), Array2::<f64>::random((20, 1))];
    let biases = [Array1::<f64>::random(20), Array1::<f64>::random(1)];

    let result = neural_network(input_data, weights, biases);
    println!("{:?}", result);
}
```

# Advantages

- The use of ndarray in Rust allows for optimized numerical operations in the neural network calculations, resulting in better performance compared to the Python code.
- The usage of ownership and borrowing semantics throughout the code, such as borrowing the weights and biases arrays, which prevents data races and memory errors.
- Rust, especially when used in combination with libraries like Rayon, allows for efficient parallelism in these calculations, taking advantage of multiple CPU cores to improve performance.

# k-Means Clustering

## Python Implementation

```
import numpy as np
from sklearn.cluster import KMeans
import random

# Generate random data similar to Rust example
data = np.array([[random.random() * 10, random.random() * 10] for _ in range(1000)])
k = 3

# Perform k-means clustering
kmeans = KMeans(n_clusters=k)
kmeans.fit(data)

# Get cluster labels and centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

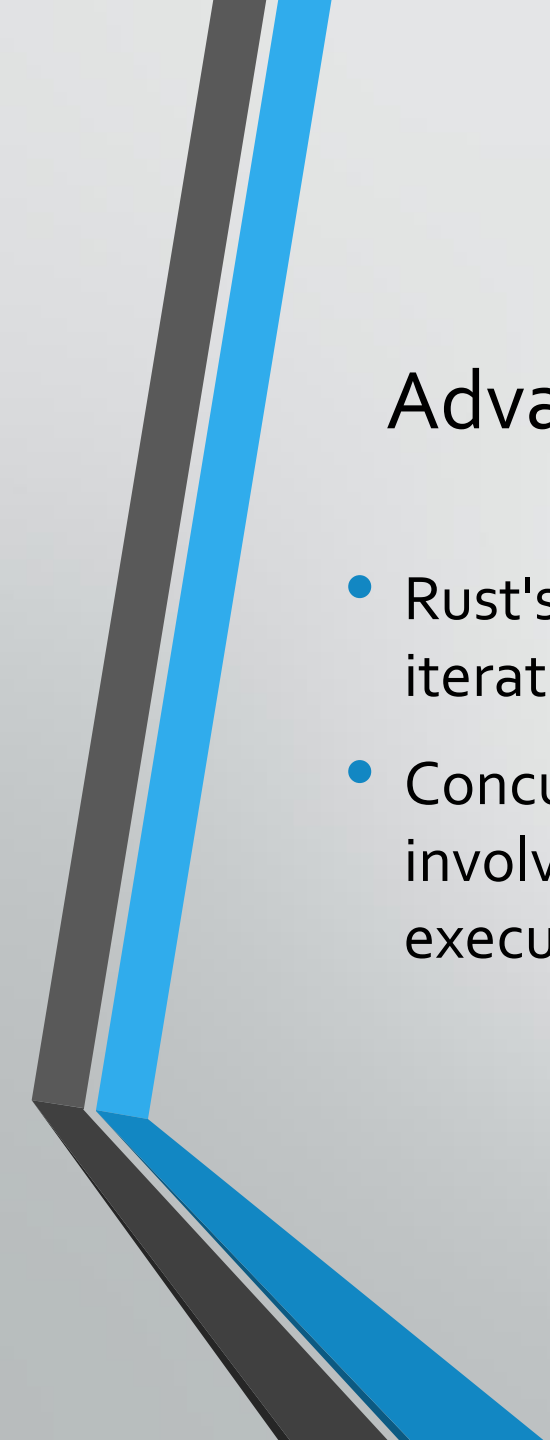
print(labels)
print(centroids)
```

## Rust Implementation

```
extern crate kmeans;
fn main() {
    let data = kmeans::generate_random_data(1000, 2);
    let k = 3;

    let (labels, centroids) = kmeans::kmeans_clustering(&data, k);

    println!("{:?}", labels);
    println!("{:?}", centroids);
}
```



## Advantages

- Rust's ownership system ensures efficient memory management during the iterative steps of the k-means clustering algorithm.
- Concurrency features in Rust can be leveraged to parallelize computations involved in calculating cluster centroids, potentially leading to faster execution

# Fast Fourier Transform (FFT)

## Python Implementation

```
import numpy as np
from numpy.fft import fft

# Generate random signal
signal = np.random.rand(1024)

# Perform FFT
fft_result = np.fft.fft(signal)

print(fft_result)
```

## Rust Implementation

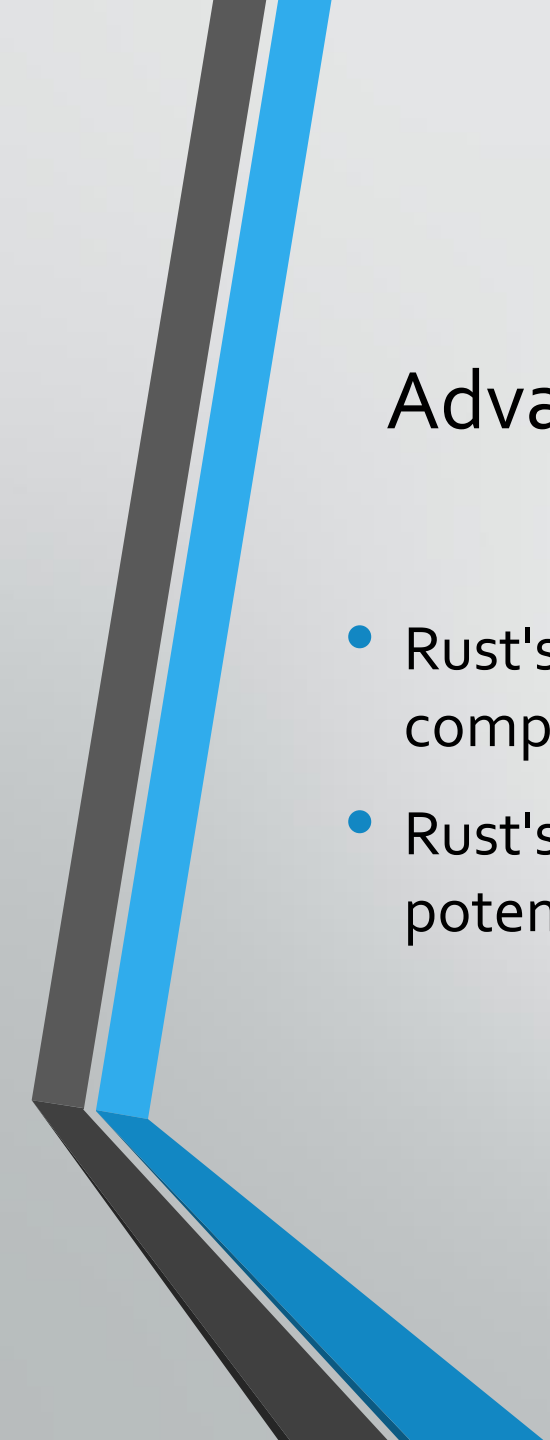
```
extern crate rustfft;

fn main() {
    let signal = rustfft::generate_random_signal(1024);

    let fft_result = rustfft::perform_fft(&signal);

    println!("{:?}", fft_result);
}
```





## Advantages

- Rust's zero-cost abstractions and low-level control can optimize FFT computations.
- Rust's concurrency features can be explored to parallelize FFT calculations, potentially leading to better performance.

# Advantages of RUST

## Ownership

- It ensures only one variable "owns" a piece of data at a time.
- No garbage collection, reducing runtime overhead.
- Memory is freed as soon as the owning variable goes out of scope.
- Comparison:
  - In Python, reference counting and garbage collection introduce runtime overhead.
  - Rust's ownership eliminates this by managing memory strictly at compile-time.

# Immutability

- Variables are immutable by default; mutability must be explicitly declared.
- Reduces complexity and makes code more predictable.
- Facilitates parallelism and optimization by the compiler.
- Comparison:
  - Python's default mutability can lead to unintended side effects and make optimization challenging.
  - Rust's default immutability enhances code clarity and allows for better compiler optimization.

# Memory Safety

- borrowing rules.
- The borrow checker enforces rules at compile-time, preventing runtime errors.
- Eliminates common sources of bugs and enhances robustness.
- Comparison:
  - Python is susceptible to runtime errors like null pointer dereferences.
  - Rust's borrow checker catches these issues at compile-time, ensuring safer and more reliable code execution.

# Borrow Checker

- Ensures that references to data do not outlive the data they point to.
- Guarantees safe concurrent programming without data races.
- Facilitates writing high-performance, concurrent code with confidence.
- Comparison:
  - Python may face data race issues due to the GIL.
  - Rust's borrow checker prevents such issues, allowing for safe and efficient concurrent programming.

# Lifetime

- Lifetimes specify the scope for which references are valid.
- Helps prevent dangling references and enhances clarity in code.
- The Rust compiler ensures that references adhere to specified lifetimes.
- Comparison:
  - Python lacks explicit lifetime annotations, which can lead to potential issues with dangling references.
  - Rust's explicit lifetime annotations ensure clarity and prevent such problems.

# Scoping

- Rust's scoping rules govern the visibility and lifespan of variables.
- Local variables' lifetimes are clearly defined, aiding in predictability.
- Scoping aligns with the ownership system, reinforcing memory safety.
- Comparison:
  - Python's scoping is less strict, potentially leading to confusion and unintended variable interactions.
  - Rust's strict scoping aligns with its ownership system, enhancing predictability and memory safety.



# FUTURE GOALS

## MILESTONE 2

- Using rust's concurrency features to improve neural network performance.
- Testing neural network performance for rust on other datasets.