

Peroxide Guide

Arect

2019-03-07

Contents

1	Prerequisites	5
2	Quick Start	7
2.1	Cargo.toml	7
2.2	Import all at once	7
3	Vector	9
3.1	Print <code>Vec<f64></code>	9
3.2	Syntactic sugar for <code>Vec<f64></code>	9
3.3	From ranges to Vector	9
3.4	Vector Operation	10
3.5	Concatenation	10
4	Matrix	13
4.1	Declare matrix	13
4.2	Basic Method for Matrix	14
4.3	Read & Write	15
4.4	Concatenation	15
4.5	Matrix operations	16
4.6	Extract & modify components	16
4.7	Conversion between vector	17
4.8	Useful constructor	17
5	Linear Algebra	19
5.1	Transpose	19
5.2	LU Decomposition	19
5.3	Determinant	20
5.4	Inverse matrix	20
5.5	Moore-Penrose Pseudo Inverse	20
6	Functional Programming	23
6.1	FP for Vector	23
6.2	FP for Matrix	25

Chapter 1

Prerequisites

- Rust
- Peroxide

Chapter 2

Quick Start

2.1 Cargo.toml

- To use `peroxide`, you should edit `Cargo.toml`
 - Current document version is corresponding to 0.8
- ```
peroxide = "0.8"
```

### 2.2 Import all at once

- You can import all functions & structures at once

```
extern crate peroxide;
use peroxide::*;

fn main() {
 //Some codes...
}
```





# Chapter 3

## Vector

### 3.1 Print Vec<f64>

- There are two ways to print vector
  - Original way: `print!("{:?}", a);`
  - Peroxide way: `a.print();` - Round-off to fourth digit

```
fn main() {
 let a = vec![2f64.sqrt()];
 a.print(); // [1.4142]
}
```

### 3.2 Syntactic sugar for Vec<f64>

- There is useful macro for Vec<f64>
- For R, there is `c`

```
R
a = c(1,2,3,4)
```

- For Peroxide, there is `c!`

```
// Rust
fn main() {
 let a = c!(1,2,3,4);
}
```

### 3.3 From ranges to Vector

- For R, there is `seq` to declare sequence.

```
R
a = seq(1, 4, 1)
print(a)
[1] 1 2 3 4
```

- For peroxide, there is `seq` to declare sequence.

```
fn main() {
 let a = seq(1, 4, 1);
 a.print();
 // [1, 2, 3, 4]
}
```

### 3.4 Vector Operation

- There are some vector-wise operations
  - `add(&self, other: Vec<f64>) -> Vec<f64>`
  - `sub(&self, other: Vec<f64>) -> Vec<f64>`
  - `mul(&self, other: Vec<f64>) -> Vec<f64>`
  - `div(&self, other: Vec<f64>) -> Vec<f64>`
  - `dot(&self, other: Vec<f64>) -> f64`
  - `norm(&self) -> f64`

```
fn main() {
 let a = c!(1,2,3,4);
 let b = c!(4,3,2,1);

 a.add(&b).print();
 a.sub(&b).print();
 a.mul(&b).print();
 a.div(&b).print();
 a.dot(&b).print();
 a.norm().print();

 // [5, 5, 5, 5]
 // [-3, -1, 1, 3]
 // [4, 6, 6, 4]
 // [0.25, 0.6667, 1.5, 4]
 // 20
 // 5.477225575051661 // sqrt(30)
}
```

- And there are some useful operations too.
  - `pow(&self, usize) -> Vec<f64>`
  - `powf(&self, f64) -> Vec<f64>`
  - `sqrt(&self) -> Vec<f64>`

```
fn main() {
 let a = c!(1,2,3,4);

 a.pow(2).print();
 a.powf(0.5).print();
 a.sqrt().print();
 // [1, 4, 9, 16]
 // [1, 1.4142, 1.7321, 2]
 // [1, 1.4142, 1.7321, 2]
}
```

### 3.5 Concatenation

There are two concatenation operations.

- `cat(T, Vec<T>) -> Vec<f64>`
- `concat(Vec<T>, Vec<T>) -> Vec<T>`

```
fn main() {
 let a = c!(1,2,3,4);
 cat(0f64, a.clone()).print();
 // [0, 1, 2, 3, 4]

 let b = c!(5,6,7,8);
 concat(a, b).print();
 // [1, 2, 3, 4, 5, 6, 7, 8]
}
```



# Chapter 4

## Matrix

### 4.1 Declare matrix

- You can declare matrix by various ways.
  - R's way - Default
  - MATLAB's way
  - Python's way
  - Other macro

#### 4.1.1 R's way

- Description: Same as R - `matrix(Vector, Row, Col, Shape)`
- Type: `matrix(Vec<T>, usize, usize, Shape)` where `T: std::convert::Into<f64> + Copy`
  - Shape: Enum for matrix shape - Row & Col

```
fn main() {
 let a = matrix(c!(1,2,3,4), 2, 2, Row);
 a.print();
 // c[0] c[1]
 // r[0] 1 2
 // r[1] 3 4

 let b = matrix(c!(1,2,3,4), 2, 2, Col);
 b.print();
 // c[0] c[1]
 // r[0] 1 3
 // r[1] 2 4
}
```

#### 4.1.2 MATLAB's way

- Description: Similar to MATLAB (But should use `&str`)
- Type: `ml_matrix(&str)`

```
fn main() {
 let a = ml_matrix("1 2; 3 4");
 a.print();
 // c[0] c[1]
 // r[0] 1 2
```

```

 // r[1] 3 4
}

```

### 4.1.3 Python's way

- Description: Declare matrix as vector of vectors.
- Type: `py_matrix(Vec<Vec<T>>)` where `T: std::convert::Into<f64> + Copy`

```

fn main() {
 let a = py_matrix(vec![vec![1, 2], vec![3, 4]]);
 a.print();
 // c[0] c[1]
 // r[0] 1 2
 // r[1] 3 4
}

```

### 4.1.4 Other macro

- Description: R-like macro to declare matrix
- For R,

```

R
a = matrix(1:4;1, 2, 2, Row)
print(a)
[,1] [,2]
[1,] 1 2
[2,] 3 4

```

- For Peroxide,

```

fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 a.print();
 // c[0] c[1]
 // r[0] 1 2
 // r[1] 3 4
}

```

## 4.2 Basic Method for Matrix

There are some useful methods for Matrix

- `row(&self, index: usize) -> Vec<f64>`: Extract specific row as `Vec<f64>`
- `col(&self, index: usize) -> Vec<f64>`: Extract specific column as `Vec<f64>`
- `diag(&self) -> Vec<f64>`: Extract diagonal components as `Vec<f64>`
- `swap(&self, usize, usize, Shape) -> Matrix`: Swap two rows or columns
- `subs_col(&mut self, usize, Vec<f64>)`: Substitute column with `Vec<f64>`
- `subs_row(&mut self, usize, Vec<f64>)`: Substitute row with `Vec<f64>`

```

fn main() {
 let a = ml_matrix("1 2; 3 4");
}

```

```

a.row(0).print(); // [1, 2]
a.col(0).print(); // [1, 3]
a.diag().print(); // [1, 4]
a.swap(0, 1, Row).print();
// c[0] c[1]
// r[0] 3 4
// r[1] 1 2

let mut b = ml_matrix("1 2;3 4");
b.subs_col(0, c!(5, 6));
b.subs_row(1, c!(7, 8));
b.print();
// c[0] c[1]
// r[0] 5 2
// r[1] 7 8
}

```

## 4.3 Read & Write

In peroxide, we can write matrix to csv.

- `write(&self, file_path: &str):` Write matrix to csv
- `write_with_header(&self, file_path, header: Vec<&str>):` Write with header

```

fn main() {
 let a = ml_matrix("1 2;3 4");
 a.write("matrix.csv").expect("Can't write file");

 let b = ml_matrix("1 2; 3 4; 5 6");
 b.write_with_header("header.csv", vec!["odd", "even"])
 .expect("Can't write header file");
}

```

Also, you can read matrix from csv.

- Type: `read(&str, bool, char) -> Result<Matrix, Box<Error>>`
- Description: `read(file_path, is_header, delimiter)`

```

fn main() {
 let a = read("matrix.csv", false, ',')
 .expect("Can't read matrix.csv file");
 a.print();
// c[0] c[1]
// r[0] 1 2
// r[1] 3 4
}

```

## 4.4 Concatenation

There are two options to concatenate matrices.

- `cbind:` Concatenate two matrices by column direction.
- `rbind:` Concatenate two matrices by row direction.

```
fn main() {
 let a = ml_matrix("1 2;3 4");
 let b = ml_matrix("5 6;7 8");

 cbind(a.clone(), b.clone()).print();
 // c[0] c[1] c[2] c[3]
 // r[0] 1 2 5 7
 // r[1] 3 4 6 8

 rbind(a, b).print();
 // c[0] c[1]
 // r[0] 1 2
 // r[1] 3 4
 // r[2] 5 6
 // r[3] 7 8
}
```

## 4.5 Matrix operations

- In peroxide, can use basic operations between matrices. I'll show you by examples.

```
fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 (a.clone() + 1).print(); // -, *, / are also available
 // c[0] c[1]
 // r[0] 2 3
 // r[1] 4 5

 let b = matrix!(5;8;1, 2, 2, Row);
 (a.clone() + b.clone()).print(); // -, *, / are also available
 // c[0] c[1]
 // r[0] 6 8
 // r[1] 10 12

 (a.clone() % b.clone()).print(); // Matrix multiplication
 // c[0] c[1]
 // r[0] 19 22
 // r[1] 43 50
}
```

## 4.6 Extract & modify components

- In peroxide, matrix data is saved as linear structure.
- But you can use two-dimensional index to extract or modify components.

```
fn main() {
 let mut a = matrix!(1;4;1, 2, 2, Row);
 a[(0,0)].print(); // 1
 a[(0,0)] = 2f64; // Modify component
 a.print();
 // c[0] c[1]
 // r[0] 2 2
}
```



```

 // r[1] 3 4
}

```

## 4.7 Conversion between vector

### 4.7.1 Vector to Matrix

- `to_matrix` method allows conversion from vector to column matrix.

```

fn main() {
 let a = c!(1,2,3,4);
 a.to_matrix().print();
 // c[0]
 // r[0] 1
 // r[1] 2
 // r[2] 3
 // r[3] 4
}

```

### 4.7.2 Matrix to Vector

- Just use `row` or `col` method (I already showed at Basic method section).

```

fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 a.row(0).print(); // [1, 2]
}

```

## 4.8 Useful constructor

- `zeros(usize, usize)`: Construct matrix which elements are all zero
- `eye(usize)`: Identity matrix
- `rand(usize, usize)`: Construct random uniform matrix (from 0 to 1)

```

fn main() {
 let a = zeros(2, 2);
 assert_eq!(a, ml_matrix("0 0;0 0"));

 let b = eye(2);
 assert_eq!(b, ml_matrix("1 0;0 1"));

 let c = rand(2, 2);
 c.print(); // Random 2x2 matrix
}

```



## Chapter 5

# Linear Algebra

### 5.1 Transpose

- Caution: Transpose does not consume the original value.

```
fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 a.transpose().print();
 // Or you can use shorter one
 a.t().print();
 // c[0] c[1]
 // r[0] 1 3
 // r[1] 2 4
}
```

### 5.2 LU Decomposition

- Peroxide uses **complete pivoting** for LU decomposition - Very stable
- Since there are lots of causes to generate error, you should use `Option`
- `lu` returns `Option<PQLU>`
  - PQLU has four field - `p`, `q`, `l`, `u`
  - `p` means row permutations
  - `q` means column permutations
  - `l` means lower triangular matrix
  - `u` means upper triangular matrix
- The structure of PQLU is as follows:

```
#[derive(Debug, Clone)]
pub struct PQLU {
 pub p: Perms,
 pub q: Perms,
 pub l: Matrix,
 pub u: Matrix,
}

pub type Perms = Vec<usize, usize>;
```

- Example of LU decomposition:

```
fn main() {
 let a = matrix(c!(1,2,3,4), 2, 2, Row);
 let pqlu = a.lu().unwrap(); // unwrap because of Option
 let (p,q,l,u) = (pqlu.p, pqlu.q, pqlu.l, pqlu.u);
 assert_eq!(p, vec![(0,1)]); // swap 0 & 1 (Row)
 assert_eq!(q, vec![(0,1)]); // swap 0 & 1 (Col)
 assert_eq!(l, matrix(c!(1,0,0.5,1),2,2,Row));
 // c[0] c[1]
 // r[0] 1 0
 // r[1] 0.5 1
 assert_eq!(u, matrix(c!(4,3,0,-0.5),2,2,Row));
 // c[0] c[1]
 // r[0] 4 3
 // r[1] 0 -0.5
}
```

### 5.3 Determinant

- Peroxide uses LU decomposition to obtain determinant ( $\mathcal{O}(n^3)$ )

```
fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 assert_eq!(a.det(), -2f64);
}
```

### 5.4 Inverse matrix

- Peroxide uses LU decomposition to obtain inverse matrix.
- It needs two sub functions - `inv_l`, `inv_u`
  - For inverse of L, U, I use block partitioning. For example, for lower triangular matrix :

$$L = \begin{pmatrix} L_1 & \mathbf{0} \\ L_2 & L_3 \end{pmatrix}$$

$$L^{-1} = \begin{pmatrix} L_1^{-1} & \mathbf{0} \\ -L_3^{-1}L_2L_1^{-1} & L_3^{-1} \end{pmatrix}$$

```
fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 a.inv().unwrap().print();
 // c[0] c[1]
 // r[0] -2 1
 // r[1] 1.5 -0.5
}
```

### 5.5 Moore-Penrose Pseudo Inverse

- $X^\dagger = (X^T X)^{-1} X$

```
fn main() {
 let a = matrix!(1;4;1, 2, 2, Row);
 let pinv_a = a.pseudo_inv().unwrap();
}
```

```
let inv_a = a.inv().unwrap();

assert_eq!(inv_a, pinv_a); // Nearly equal (not actually equal)
}

// PartialEq implements
impl PartialEq for Matrix {
 fn eq(&self, other: &Matrix) -> bool {
 if self.shape == other.shape {
 self.data.clone()
 .into_iter()
 .zip(other.data.clone())
 .all(|(x, y)| nearly_eq(x,y)) && self.row == other.row
 } else {
 self.eq(&other.change_shape())
 }
 }
}
```



## Chapter 6

# Functional Programming

### 6.1 FP for Vector

- There are some functional programming tools for `Vec<f64>`

```
pub trait FPVector {
 type Scalar;

 fn fmap<F>(&self, f: F) -> Self
 where
 F: Fn(Self::Scalar) -> Self::Scalar;
 fn reduce<F, T>(&self, init: T, f: F) -> Self::Scalar
 where
 F: Fn(Self::Scalar, Self::Scalar) -> Self::Scalar,
 T: convert::Into<Self::Scalar>;
 fn zip_with<F>(&self, f: F, other: &Self) -> Self
 where
 F: Fn(Self::Scalar, Self::Scalar) -> Self::Scalar;
 fn filter<F>(&self, f: F) -> Self
 where
 F: Fn(Self::Scalar) -> bool;
 fn take(&self, n: usize) -> Self;
 fn skip(&self, n: usize) -> Self;
}
```

#### 6.1.1 fmap

- `fmap` is syntactic sugar for `map`

```
fn main() {
 let a = c!(1,2,3,4);

 // Original rust
 a.clone()
 .into_iter()
 .map(|x| x + 1f64)
 .collect::<Vec<f64>>()
 .print();
 // [2, 3, 4, 5]
}
```

```

 // fmap in Peroxide
 a.fmap(|x| x + 1f64).print();
 // [2, 3, 4, 5]
}

```

### 6.1.2 reduce

- reduce is syntactic sugar for fold

```

fn main() {
 let a = c!(1,2,3,4);

 // Original rust
 a.clone()
 .into_iter()
 .fold(0f64, |x, y| x + y)
 .print(); // 10

 // reduce in Peroxide
 a.reduce(0f64, |x, y| x + y).print(); // 10
}

```

### 6.1.3 zip\_with

- zip\_with is composed of zip & map

```

fn main() {
 let a = c!(1,2,3,4);
 let b = c!(5,6,7,8);

 // Original rust
 a.clone()
 .into_iter()
 .zip(&b)
 .map(|(x, y)| x + *y)
 .collect::<Vec<f64>>().print();
 // [6, 8, 10, 12]

 // zip_with in Peroxide
 a.zip_with(|x, y| x + y, &b).print();
 // [6, 8, 10, 12]
}

```

### 6.1.4 filter

- filter is just syntactic sugar for filter

```

fn main() {
 let a = c!(1,2,3,4);
 a.filter(|x| x > 2f64).print();
 // [3, 4]
}

```



### 6.1.5 take & skip

- take is syntactic sugar for take

```
fn main() {
 let a = c!(1,2,3,4);
 a.take(2).print();
 // [1, 2]
}
```

- skip is syntactic sugar for skip

```
fn main() {
 let a = c!(1,2,3,4);
 a.skip(2).print();
 // [3, 4]
}
```

## 6.2 FP for Matrix

- Similar to FPVector

```
pub trait FP {
 fn take(&self, n: usize, shape: Shape) -> Matrix;
 fn skip(&self, n: usize, shape: Shape) -> Matrix;
 fn fmap<F>(&self, f: F) -> Matrix where F: Fn(f64) -> f64;
 fn reduce<F, T>(&self, init: T, f: F) -> f64
 where F: Fn(f64, f64) -> f64,
 T: convert::Into<f64>;
 fn zip_with<F>(&self, f: F, other: &Matrix) -> Matrix
 where F: Fn(f64, f64) -> f64;
}
```

- Above functions play same roles as FPVector