Rusty Notes

Option Pricing - Monte Carlo

link: https://play.rust-lang.org/?gist=1ba216a0728252be16c45bd8d9d7b50d

Topics covered:

1. standard normal rv generation
2. math functions

```
extern crate rand;
use rand::Rng;
use rand_distr::StandardNormal;
use std::f32;

fn call(s: f32, x: f32) -> f32 {
    return f32::max(s - x, 0.0);
}

fn main() {
    let k: f32 = 1.0;
    let vol: f32 = 0.20;
    let rf: f32 = 0.03;
    let t = 250;

    let dt: f32 = 1.0 / 252.0;

    let mu = (rf - 0.5 * vol * vol) * dt;
    let sigma = vol * f32::sqrt(dt);

    let mut p: f32 = 0.0;
    for _i in 1..1000 {
        let logs0: f32 = 0.0;
        let mut logst: f32 = logs0;
        for _j in 1..=t {
            let z: f32 = rand::thread_rng().sample(StandardNormal);
            logst += mu + sigma * z;
        }
        p += call(f32::exp(logst), k);
    }
    p /= 1000.0;
    println!("price: {}", p);
}
```

Enums

Enums are special types that can take finite set of values, called variance of an enum.

Rust enums differ from enums in C/C++. They are similar to algebraic data types of Haskell.

To create an enum:

```
enum MyEnum {
    VALUE1,
    VALUE2,
}
let myvar = MyEnum::VALUE1;
```

Rust enum variants can take different types of data. For example,

```
enum MyEnum {
    VALUE1(String),
    VALUE2(u8),
}

let s = String::from("abc");
let u: u8 = 123;
let v1 = MyEnum::VALUE1(s);
let v2 = MyEnum::VALUE2(u);
```

Returning Custom Errors

link: https://play.rust-lang.org/?gist=1690e89f38dd1d19b726c3644ca04192

Topic: Result enum.

```
fn sqrt(a: f32) -> Result<f32, String> {
    if a >= 0.0 {
        return Ok(a.sqrt());
    }
    Err("negative number".to_string())
}

fn main() {
    match sqrt(-4.0) {
        Ok(x) => println!("{:?}", x),
        Err(s) => println!("{:?}", s),
    }
}
```

Guessing game

link: https://play.rust-lang.org/?gist=9c8840c989e884eb3fbab4698796fc45

Topics:

1. Reading from stdin
2. Parsing string to int
3. Random int generation
4. Comparing with cmp::Ordering
5. Match

```
extern crate rand;
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    let secret = rand::thread_rng().gen_range(1..=100);
    println!("guess the secret number:");
    loop {
        let mut guess = String::new();
        io::stdin()
            .read_line(&mut guess)
            .expect("error reading input");
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        match guess.cmp(&secret) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
        println!("guess again:");
    }
}
```

Option Enum

link: https://play.rust-lang.org/?gist=ae5963045e1d2eaf501f06ea8afe24e9

Topics: Option Enum

Option Enum for a type T is defined as follows:

```
enum Option<T> {
    Some(T),
    None,
}
```

It is a type representing two possible states - valid value or none (or null). Using this type forces checking and handling of null values.

```
fn add(a: Option<i32>, b: Option<i32>) -> Option<i32> {
    if a.is_some() && b.is_some() {
        let c : i32 = a.unwrap() + b.unwrap();
        return Some(c);
    }
    None
}

fn main() {
    let a : Option<i32> = Some(5);
    let b : Option<i32> = None;
    let c = add(a, b);
    println!("{:?}", c);
}
```

Match Expression

Match is an expression, so it can be assigned to a variable. In the example below, if parse does not return an error, then s takes the value of the result of the statement block corresponding to the Ok variant. Otherwise, it takes the result of the statement block corresponding to the Err variant, in this case 99. Like function body the last statement should be without semi-colon for it to be returned.

```
fn main() {
    let guess = "123\n".to_string();
    let guess: u32 = match guess.trim().parse() {
        Ok(num) => {
            println!("well done!");
            num
        }
        Err(_) => {
            println!("bad case");
            99
        }
    };
```

```
    println!("{}", guess)
}
```

Ownership

When a variable is passed to a function, the function takes ownership of the memory referenced by the variable.

```
#[derive(Debug)]
struct Some_struct {
    x: i32,
}
fn foo(s : Some_struct) {
    println!("{:?}", s)
}
fn main() {
    let s = Some_struct{5};
    foo(s);
    foo(s);
}
```

This results in an error because the first foo owns s and the compiler does not allow s to be moved again.

One way is to use clone(). This copies the struct when passing as argument. Clone needs to be derived like Debug for custom types.

```
#[derive(Debug, Clone)]
struct Some_struct {
    x: i32,
}
fn foo(&s : Some_struct) {
    println!("{:?}", s)
}
fn main() {
    let s = Some_struct{x: 5};
    foo(s.clone());
    foo(s);
}
```

Another way is to use Copy, which is implicit.

```
#[derive(Debug, Clone, Copy)]
struct SomeStruct {
    x: i32,
}
fn foo(s : SomeStruct) {
    println!("{:?}", s)
}
```

```
fn main() {
    let s = SomeStruct{ x: 5 };
    foo(s);
    foo(s);
}
```

Copying structs for every function call may not be performant.

Third way is to use read only reference. This is a way of handing over a variable without handing over ownership. Because it is read only, one can have as many read only references as needed. Here foo() does not modify the struct, so read only reference works.

```
#[derive(Debug)]
struct SomeStruct {
    x: i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s.x)
}
fn main() {
    let s = SomeStruct{ x: 5 };
    foo(&s);
    foo(&s);
}
```

What if the function modifies the struct passed? We use mutable reference. Rust allows only one mutable reference, however.

```
#[derive(Debug)]
struct SomeStruct {
    x: i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s.x)
}
fn bar(s: &mut SomeStruct) {
    s.x = 2;
}
fn main() {
    let s = SomeStruct{ x: 5 };
    foo(&mut s);
    foo(&s);
}
```

In the example below, s is borrowed immutable by u. Then a mutable borrow occurs in the call to bar(). However, the immutable borrow of u is still in scope, as it is used in the call to foo() after. This results in a compilation error.

```
fn main() {
    let mut s = SomeStruct{ x: 5 };
    let u: &SomeStruct = &s;
    bar(&mut s);
    foo(u);
}
```

Lifetime

Lifetime is relevant only to references. Essentially making sure reference to memory does not continue to exist, when the memory has been freed.

Example:

```
#[derive(Debug)]
struct SomeStruct {
    x: i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s.x)
}
fn max<'a>(a: &'a SomeStruct, b: &'a SomeStruct)-> &'a SomeStruct {
    if a.x > b.x {
        a
    } else {
        b
    }
}
fn main() {
    let s = SomeStruct{ x: 5 };
    let u = SomeStruct{ x: 2 };
    let t = max(&s, &u);
    foo(t);
}
```

Lifetimes are specified with a single quote and reference variable name. Here, the annotation ensures that the return reference exists as long as both input references exist.

Struct definitions also need lifetime annotation if it has reference type fields. For example,

```
#[derive(Debug)]
struct SomeStruct<'a> {
    x: &'a i32,
}
fn foo(s : &SomeStruct) {
    println!("{:?}", s)
```

```
}
fn main() {
    let n = 3;
    let s = SomeStruct{ x: &n };
    foo(&s);
}
```

Local Variables

Inside a function, local variables are not initialized when allocated. Local variables can be used only after they have been initialized.

```
fn foo() {
    let x: u8 = 0;
    let y: f32;
    println!("{}", x);
    println!("{}", y); //error[E0381]: used binding `y` isn't initialized
}
```

If is an Expression

```
let x = 5;
let y = if x == 5 { 10 } else { 15 }; // y: i32
```

Expressions return value and in this case the last expression in the final conditional.

Ranging Over Vector

A for loop takes ownership of the vector.

```
fn main() {
    let v = vec![0, 1, 2, 3, 4];
    for i in v {
        print!("{} ", i);
    }
    println!("{}", v[0]); // error because the loop above took ownership of v
}
```

The alternative is to use a reference.

```
fn main() {
    let v = vec![0, 1, 2, 3, 4];
    for i in &v {
        print!("{} ", i);
    }
    println!("{}", v[0]);
}
```

Vector

Constructor:

```
let vec = vec![1,2,3];
let vec: Vec<i32> = vec::new();
let vec = vec![0; 5]; // vector of length 5, filled with 0.
```

Attributes:

```
vec.len();
vec.is_empty();
```

Iteration: - iterating over reference to vec gives reference to each element - iterating over a moved vec gives value of each element

```
let vecs = vec![1, 2, 3, 4, 5];

// iterating over reference
let mut s = 0;
for v in &vecs {
    s += *v; // v has type &i32 and needs to be dereferenced
}
println!("{:?}", vecs);

// iterating over a moved vec
s = 0;
for v in vecs {
    s += v; // v has type i32
}
println!("{}", s);
println!("{:?}", vecs); // error vecs was moved, no invalid
```

Strings

Strings are sequences of UTf-8 bytes. There are two types: &str and String. &str (string slice) is a string literal and is statically allocated i.e. stored in the binary. &str is a reference to a sequence of UTF-8 bytes.

```
let s = "hello world";
```

String type is heap allocated and constructed from &str with to_string() method.

```
let s = "hello world".to_string();
```

String type is mutable while &str is immutable.

```
let s = "hello".to_string();
s.push_str(", world!");
```

A String will coerce into &str with a &.

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}
fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

However, coercion does not happen for functions that accept a trait of &str.

Strings can be concatenated with a +.

```
let s = "hello".to_string();
let t = ", world!";
println!("{}", s + t);

let u = ", world!".to_string();
println!("{}", s + &u); //&u coerces to &str
```

Because of UTF-8 encoding (i.e. each character can take up to 4 bytes), indexing a String is tricky. Indexing produces byte indexing.

```
// prints h e l l o
let s = "hello".to_string;
for i in s.chars() {
    print!("{} ", i);
}

// prints bytes
for i in s.as_bytes {
    print!("{}", i);
}

// gets ith character
// nth() returns an Option<&str> enum
let c = s.chars().nth(1).unwrap();
println!("{:?}", c);
```

Generics

Parametric polymorphism. Couple of frequently used examples from standard library:

```
enum Option<T> {
    Some(T),
    None,
}

enum Result<T, E> {
```

```
    Ok(T),
    Err(E),
}
```

We could use any symbol.

```
enum Result<A, B> {
    Ok(A),
    Err(B),
}
```

Generic custom type example:

```
struct Point<T> {
    x: T,
    y: T,
}

let p1 = Point{x: 0, y: 0} // Point<i32>
let p2 = Point{x: 0.0, y: 0.0} // Point<f32>
```

struct fields are private by default, even if struct is marked pub.

Generic Functions

Similar to structs.

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

The angle bracket specification is needed because the compiler needs to know T in parameter list is not a concrete type.

If Expression

If is an expression.

```
let y = if val > 0.0 {
        println!("greater than 0");
        val
    } else {
        println!("less than 0");
        val
    };
    println!("{:?}", y);
```

Ownership

Every value is owned by a variables. And there can only be one owner. For example, in the below code, value owned by x is moved to y i.e. y takes the ownership.

When the owner goes out of scope its value will be dropped. This means when a function takes ownership of a value, it must return it for it to be used subsequently.

```
let x = String::from("hello world");
let y = x; //y owns x's value i.e. x's value moved to y.
println!("{}", x); // compile error; x is invalid after the move
```

However, x's value can be "borrowed". This is called referencing. There are two types of references: mutable and immutable.

Immutable reference is read only borrow. Since ownership integrity is not affected and all other immutable borrows refer to the same value, one can have any number of immutable references.

Only one mutable reference is allowed, since it is almost like a move.

```
let x = String::from("hello world");
let y = &x; //y borrows x's value immutably.
println!("{}", y);
println!("{}", x); // valid
let z = &x; // another immutable borrow
for c in z.chars() {
    println!("{}", c);
}
println!("{}", x); // valid
```

However, below results in compile error.

```
let mut x = String::from("hello world");
let y = &mut x; //mutable borrow of x, almost like a move
let z = &x; //immutable borrow of the same x; compile error as y can change x
println!("{}", y.to_uppercase());
println!("{}", z);
```

Stack Variables

Variables that live entirely in the stack memory are copied and not moved.

```
let x = [0; 5];
let n = 5;
let s = String::from("hello");
foo(x);
println!("{:?}, {}", x, n); // works, since x and n were copied and not moved to foo.
```

```
bar(s);
println!("{}", s); // compile error, s was moved to bar.
```

Printing Type of a variable

We can use the std::any::typename function.

```
fn print_type_of<T>(_: &T) {
    println!("{}", std::any::type_name::<T>())
}
```

println! & format! & push_str

Both borrow implicitly even if values are passed without explicit referencing. Values are not moved.

```
let s = "abc".to_string();
println!("{}", s); //s is implicitly borrowed
let s2 = format!("{}{}", s, "def"); // s is implicitly borrowed
println!("{}", s); // works
let mut s3 = s; //s is moved and cannot be used after this
let s4 = "ghi";
s3.push_str(s4); // s4 is implicitly borrowed
s3.push_str(s4); // valid
```

Collections consume

Using a owned value to construct a vector, struct or hash map moves the value.

```
struct Foo {
    x: String,
    y: usize,
}

fn main() {
    let mut s = String::from("abc");
    let v = vec![s]; // s is moved to v
    s.push_str("def"); // error

    let mut s = String::from("abc");
    let mut h: HashMap<_, _> = HashMap::new();
    h.insert(s, 0); // s is moved to h
    s.push_str("def"); // error

    let mut s = String::from("abc");
    let foo = Foo { x: s, y: 0 }; // s is moved to foo
    s.push_str("def"); // error
}
```

Retrieving Values from Hash Map

The get function takes a key and returns an Option enum wrapping a reference to the value. If the key does not exist, we get None.

```
let mut h: HashMap<_, _> = HashMap::new();
h.insert("djokovic", 1);
h.insert("federer", 2);
h.insert("nadal", 3);
println!("{:?}", h);

// get returns a mutable reference
let name = "nadal";
if let Some(rank) = h.get(&name) {
    println!("{} is seeded {}", name, rank);
} else {
    println!("{} is unseeded", name);
}
```

One can also iterate a hash map like a vector.

```
for (key, val) in &h {
    println!("{}: {}", key, val);
}
```

The order of output is unknown.

Updating Hash Map

If a key alreadey existings, inserting overwrites existing value.

```
h.insert("nadal", 4); // nadal's prev value of 3 is overwritten
```

The entry method provides more flexibility. It returns an Entry enum which has a or_insert() that returns a mutable reference to existing value (if it exists) or sets the value to 0 and returns a mutable reference to it.

```
// first insert
// inserts (wawrinka, 8) in h and returns mutable ref to 8
let mut v = h.entry("wawrinka").or_insert(8);

// "wawrinka" exists, so returns mutable ref to 3
let mut v = h.entry("wawrinka").or_insert(3);

// updating based on existing value
// example: counting occurences
let text = "quick brown fox jumped over the fence";
let mut map = HashMap::new();
for w in text.split_whitespace() {
    let mut count = map.entry(w).or_insert(0);
```

```
    *count += 1;
}
```

Example: Mean, Mode, Median

```
use std::collections::HashMap;

fn main() {
    let vecs: Vec<Vec<i32>> = vec![[1, 2, 3, 4, 1].to_vec(), [1, 2, 2, 2, 1].to_vec(), [1].t

    for v in &vecs {
        println!(
            "{:?}: mean {}, median {}, mode {}",
            v,
            mean(&v),
            median(&v),
            mode(&v)
        );
    }
}

fn mean(x: &Vec<i32>) -> f32 {
    let mut s: f32 = 0.0;
    for v in x {
        s += *v as f32;
    }
    s / x.len() as f32
}

fn median(x: &Vec<i32>) -> f32 {
    let mut y = x.clone();
    y.sort();
    if y.len() % 2 == 0 {
        let i = (y.len() / 2) as usize;
        (y[i - 1] + y[i]) as f32 * 0.5
    } else {
        let i = (y.len() / 2) as usize;
        y[i] as f32
    }
}

fn mode(x: &Vec<i32>) -> i32 {
    let mut h = HashMap::new();
    for v in x {
        let count = h.entry(v).or_insert(0);
        *count += 1;
```

```
    }
    let mut maxcount = 0;
    let mut modenum = x[0];
    for (key, val) in h {
        if val > maxcount {
            maxcount = val;
            modenum = *key;
        }
    }
    modenum
}
```

String Manipulation Example

```
const VOWELS: [char; 5] = ['a','e','i','o','u'];

fn to_pig_latin(s: &str) -> String {
    let c: Vec<char> = s.chars().collect();
    let tmp: String;
    let mut suffix = "-".to_string();

    if VOWELS.iter().any(|x| {x == &c[0]}) {
        tmp = c.iter().collect();
        suffix.push_str("hay");
    } else {
        tmp = c[1..].iter().collect();
        suffix.push(c[0]);
        suffix.push_str("ay");
    }
    tmp + &suffix
}
```

The ? operator

The ? operator works the same way match expression on Result enum. If the value of Result is Ok, then the value wrapped by Ok will be returned from the expression and program continues. If it is Err, then Err will be returned from the whole function as if we had used return keyword.

In otherwords, ? help save a lot of boiler plat code.

```
fn read_file(f: &str) -> Result<(),Box<dyn Error>> {
    let contents = fs::read_to_string(f)?;
    println!("{}", contents);
    ...
    ...
```

```
    Ok(())
}
```

Closures

Fibonacchi example:

```
fn fib() -> Box<dyn FnMut() -> i64> {
    let mut x0: i64 = 0;
    let mut x1: i64 = 1;
    Box::new(move || {
        let z = x0 + x1;
        x0 = x1;
        x1 = z;
        z
    })
}
```

Iterators

iter borrows immutably, iter_mut borrows mutably and into_iter takes ownership.

```
let v = vec![1, 2, 3, 4, 5];

//mut is required for next() as next consumes
let mut v1 = v.iter(); // v is borrowed
println!("{:?}", v1);

v1.next();
println!("{:?}", v1); // v1 is being consumed

println!("{:?}", v); // v is still valid

let mut v2 = v.into_iter(); // v is moved
println!("{:?}", v2);

v2.next();
println!("{:?}", v2); // v2 is being consumed

println!("{:?}", v); // error v was moved

let mut v = vec![1, 2, 3, 4, 5];
let v3 = v.iter_mut(); // v is borrowed mutably
println!("{:?}", v3);

v3.for_each(|x| *x += 7); //x is &mut to each element of v
```

```
// v.iter_mut().for_each(|x| *x += 7);
println!("{:?}", v); // v is still valid as it was borrowed, but changed
```

Iterators are lazy. Need to be consumed to force evaluation.

```
let mut numbers = vec![1, 2, 3, 4];
    for x in numbers.iter_mut() {
        *x *= 3;
    }
    println!("{:?}", numbers);
```

Same code implemented with iterator. `for_each` is preferred for side effects as it is also a consumer.

```
    numbers
        .iter_mut() // produces mutable reference to each element
        .for_each(|x| *x *=3);
    println!("{:?}", numbers);
```

Custom Box<> and Deref Implementation

```
use std::ops::Deref;

#[derive(Debug)]
struct MyBox<T>(T); // tuple struct with one elem of type T

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}

fn main() {
    let x = 3;
    let y = MyBox::new(x);
    println!("{:?}", y); // MyBox(3)
    println!("{}", *y); // 3
    println!("{}", y.deref()); // 3
    type_of(&y); // MyBox<i32>
    type_of(y.deref()); // i32
```

```
    let name = MyBox::new("balaji");
    greet(&name); // hello balaji!

    let name = MyBox::new(String::from("balaji"));
    greet(&name); // hello balaji!
}

fn greet(s: &str) {
    println!("hello {}!", s);
}

fn type_of<T>(_: &T) {
    println!("{}", std::any::type_name::<T>())
}
```

Custom Drop for Smart Pointers

Drop trait lets us customizewhat happens when a value goes out of scope. Drop trait requires implementation of a drop method that takes a mutable reference to self.

```
#[derive(Debug)]
struct MySmartPointer {
    data: String,
}
// Drop is included in the prelude
impl Drop for MySmartPointer {
    fn drop(&mut self) {
        println!("{} is dropped", self.data);
    }
}

fn main() {
    let x = MySmartPointer{ data: String::from("balaji") };
    {
        let y = MySmartPointer{ data: String::from("lauriane") };
        type_of(&y);
    }
    type_of(&x);
}

fn type_of<T>(_: &T) {
    println!("{}", std::any::type_name::<T>())
}
```

Creating a Tree Datastructure without Circular Reference

Weak references don't express an ownership relationship, unlike Rc. Therefore they are useful to create self-referential datastructures without risk of reference cycles.

```
use std::rc::{Rc, Weak};
use std::cell::RefCell;

#[derive(Debug)]
#[allow(dead_code)]
struct Node {
    value: i32,
    parent: RefCell<Weak<Node>>,
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node {
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });
    println!("leaf strong = {}, weak = {}", Rc::strong_count(&leaf), Rc::weak_count(&leaf));
    {
        let branch = Rc::new(Node {
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });
        *leaf.parent.borrow_mut() = Rc::downgrade(&branch);
        println!("branch strong = {}, weak = {}", Rc::strong_count(&branch), Rc::weak_count(
        println!("leaf strong = {}, weak = {}", Rc::strong_count(&leaf), Rc::weak_count(&lea
    }
    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
    println!("leaf strong = {}, weak = {}", Rc::strong_count(&leaf), Rc::weak_count(&leaf));
}
```

Random Number Generation

Rng trait has standard methods. thread_rng() gives a thread-local generator implementing Rng.

```
use rand::{Rng, thread_rng};
use rand_distr::StandardNormal; // 0.4.3
use rand::prelude::{IteratorRandom, SliceRandom}; // trait for iterators and vectors
```

```rust
let mut rng = thread_rng(); // generator

// fill pre alloctaed array with uniform variates
let mut x = [0.0f32; 10];
rng.fill(&mut x);

// fill pre-allocated array with u8
let mut x = [0u8; 10];
rng.fill(&mut x);

// generate an int within a range
let r: i32 = rng.gen_range(-10..=10);

// generate a rand based on receiver type
let r: i32 = rng.gen(); // int 32
let r: f32 = rng.gen(); // uniform in (0,1)

    // shuffle array
let mut a = [1,2,3,4,5];
a.shuffle(&mut rng);

// shuffle a vector
let mut v: Vec<u8> = (0..10).collect();
v.shuffle(&mut rng); // Vec<> implements SliceRandom trait

// byte slice
let mut bytes = "Hello, random!".to_string().into_bytes();
bytes.shuffle(&mut rng);
let str = String::from_utf8(bytes).unwrap();

// char slice, valid utf-8
let mut c: Vec<char> = String::from("hello world").chars().collect(); // into utf-8 chars
c.shuffle(&mut rng);
let s: String = c.iter().collect(); // back to string

// iterators
let faces = "abcd";
if let Some(c) = faces.chars().choose(&mut rng) {
    // returns a single element wrapped as Option
    println!("I am {}!", c);
};
let c = faces.chars().choose(&mut rng).unwrap();
let c = faces.chars().choose_multiple(&mut rng, 3); // vector

// generate from a distribution
use rand_distr::StandardNormal;
```

```rust
let z: f32 = rng.sample(StandardNormal);
```

Multi-variate Normal Generation

```rust
use std::f32;
use rand::rngs::ThreadRng;

struct MVNormal {
    chol: Vec<Vec<f32>>,
}

impl MVNormal {
    fn new(a: Vec<Vec<f32>>) -> MVNormal {
        let mut g = a.clone();
        for j in 0..a.len() {
            let mut v = a[j].clone();
            if j > 0 {
                for k in 0..j {
                    for l in 0..v.len() {
                        let n: usize = g[k].len()-v.len();
                        let tmp = &g[k][n..];
                        v[l] = v[l] - tmp[0] * tmp[l];
                    }
                }
            }
            for l in 0..v.len() {
                g[j][l] = v[l]/f32::sqrt(v[0]);
            }
        }
        MVNormal{ chol: g }
    }

    fn gen(&self, rng: &mut ThreadRng) -> Vec<f32> {
        let mut z: Vec<f32> = Vec::new();
        for _i in 0..self.chol.len() {
            z.push(rng.sample(StandardNormal));
        }
        let mut w = z.clone();
        for i in 0..2 {
            w[i] = 0.0;
            for j in 0..i + 1 {
                w[i] += self.chol[i][j] * z[j]
            }
        }
        w
    }
```

```
}
```

Concurrency

Threads

Rust model is 1:1 i.e. one programming thread per operating system thread.

We pass a closure containing code to be executed in a thread to thread::spawn function. The function returns a handle on which its join method can be called to ensure its completion by blocking the current thread.

```
use std::thread;
use std::time::Duration;

fn main() {
    let s = String::from("hello");
    let h = thread::spawn(move||{
    // move annotation makes the closure take ownership of s
        print!("{}", s);
        thread::sleep(Duration::from_millis(1))
    });
    h.join().unwrap(); // ensures h returns here
    println!(" world");
    // prints "hello world"
}
```

Channels

```
use std::thread;
use std::sync::mpsc; // multi producer, single customer
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel(); // create a channel
    thread::spawn(move ||{ // move neede to take ownership of tx
        let s = String::from("love");
        tx.send(s).unwrap(); // returns a Result<,>
        thread::sleep(Duration::from_millis(1))
    });

    let received = rx.recv().unwrap(); // returns Result<,>
    println!("received {}", received);
}
```

Crossbeam Channels

Crossbeam is recommended over std::sync::mpsc.

Two types of channels - unbounded and bounded. Bounded is similar to buffered channels of golang. Unbounded channels have infinite channel capacity.

```
use crossbeam::channel::{self, Receiver, Sender};
use std::{thread, time::Duration};
```

Create channels for example:

```
 let (task_tx, task_rx) = channel::unbounded();
```

Clone ends of a channel as required. For example, if we want two workers to receive on a channel, clone the receiver ends.

```
let task_rx2 = task_rx.clone();
```

An example. First define a worker function that receives tasks and sends output.

```
fn worker(name: &str, task: Receiver<u8>, result: Sender<u8>) {
    for t in task {
        println!("Worker {name} received task {t}");
        thread::sleep(Duration::from_secs_f32(0.5f32));
        println!("Worker {name} completed task {t}");
        if result.send(t).is_err() {
            break;
        }
    }
}
```

We use two workers. So two task receiver channels and two result sender channels.

```
let (task_tx, task_rx) = channel::unbounded();
let task_rx2 = task_rx.clone();
let (res_tx, res_rx) = channel::unbounded();
let res_tx2 = res_tx.clone();
```

Create workers i.e. two threads and pass the executing code as closure.

```
// create workers
let bob = thread::spawn(|| worker("bob", task_rx2, res_tx));
let rob = thread::spawn(|| worker("rob", task_rx, res_tx2));
```

Send tasks over the task channel and close the channel after. Closing the channel does not affect the items already in the channel. The receiver side for loop will exit if the channel is closed.

```
for t in 0..5 {
    task_tx.send(t).unwrap();
}
drop(task_tx);
```

Collect result and terminate threads.

```
for r in res_rx {
    println!("received {r}");
}
bob.join().unwrap();
rob.join().unwrap();
```

Errors - custom implementation

Use Enums for defining errors. Group errors into enums, as small a collection as possible.

Do not pass through errors from third party dependencies. Changes in that library may break the package.

Error should be non-exhaustive.

```
#[non_exhaustive]
pub enum MyError {
    Err1,
    Err2,

}
```

`non-exhaustive` does not allow matching without wild card. This allows adding new variants in the future.

Error trait is sub trait of Debug and Display, so these need to be implemented before.

Debug trait can simply be derived with `#[derive(Debug)]`.

To implement Display trait, all that is needed is to implement `fmt` function that takes the error and creates a string.

```
use std::fmt::{Display, Formatter};
impl Display for MyError {
    fn fmt(&self, f: &mut Formatter) -> std::fmt::Result {
        use MyError::*;
        match self {
            Err1 => write!(fmt, "error 1"),
            Err2 => write!(fmt, "error 2"),
        }
    }
}
```

And then the Error trait itself:

```
use std::error::Error;
impl Error for MyError {}
```

Errors - thiserror crate

The implementation of Display and Error traits can be wrapped into a single step with thiserror third party library.

```
use thiserror::Error;

#[derive(Debug, Error)]
#[non_exhaustive]
pub enum MyError {
    #[error("error 1")]
    Err1,
    #[error("error 2")]
    Err2,
}
```