

# Vectorized Code

Song Liu ([song.liu@bristol.ac.uk](mailto:song.liu@bristol.ac.uk))

GA 18, Fry Building,

Microsoft Teams (search "song liu").

# Code Vectorization

- R comes with many efficient operations that applies on vectors/matrices directly.
  - These operations will be translated into SIMD instructions to accelerate computation.
- Code using vector operations/functions instead of scalar operations and `for` loops is called **vectorized code**.
- Vectorized R code runs significantly faster than non-vectorized R code.

# Different Levels of Vectorization

Vectorization can happen at different "levels". Let us demonstrate this using the matrix multiplication.

# Different Levels of Vectorization

- Non-vectorized matrix multiplication in R using scalar operations only

```
matmul1 <- function(A,B){  
  # Create a zero matrix C  
  C <- matrix(0, nrow = dim(A)[1], ncol = dim(B)[2])  
  # Loop over rows of A  
  for (i in 1:dim(A)[1]){  
    # Loop over cols of B  
    for (j in 1:dim(B)[2]){  
      # Loop over cols of A  
      for (k in 1:dim(A)[2]){  
        C[i,j] <- C[i,j] + A[i,k]*B[k,j]  
      }  
    }  
  }  
  return(C)  
}
```

# Different Levels of Vectorization

- Using vector ops to replace the inner most `for` loop.

```
matmul2 <- function(A,B){  
  # Create a zero matrix C  
  C <- matrix(0, nrow = dim(A)[1], ncol = dim(B)[2])  
  # Loop over rows of A  
  for (i in 1:dim(A)[1]){  
    # Loop over rows of B  
    for (j in 1:dim(B)[2]){  
      C[i,j] <- A[i,] %*% B[, j]  
    }  
  }  
  return(C)  
}
```

$$\circ C_{i,j} = A_{[i,\cdot]} \cdot B_{[\cdot,j]}$$

# Different Levels of Vectorization

- Using vector ops to replace the **two** innermost **for** loop.

```
matmul3 <- function(A,B){  
  # Create a zero matrix C  
  C <- matrix(0, nrow = dim(A)[1], ncol = dim(B)[2])  
  # Loop over rows of A  
  for (i in 1:dim(A)[1]){  
    C[i, ] <- A[i, ]%*%B  
  }  
  return(C)  
}
```

- $C_{[i,\cdot]} = A_{[i,\cdot]} \cdot B$

# Different Levels of Vectorization

- Using built-in R `%*%` function with no `for` loop: `A%*%B`

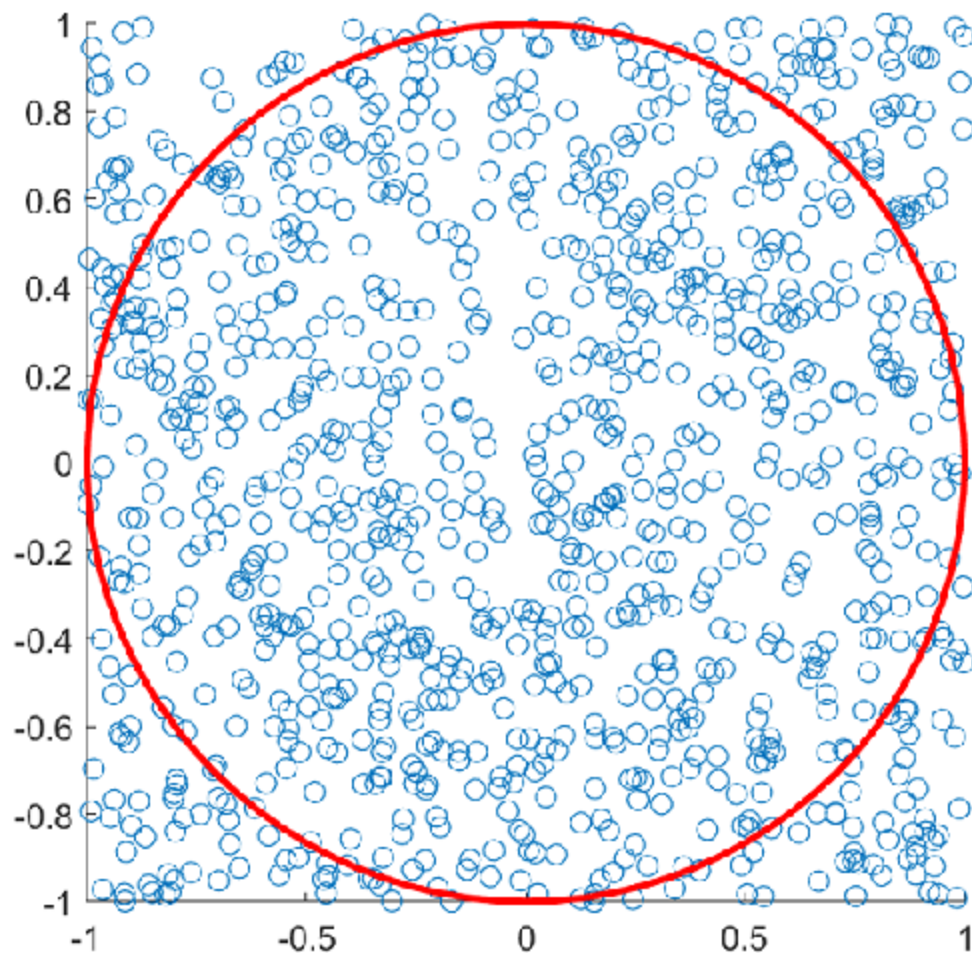
# Performance of Vectorized Code

- Let us test the performance of these implementations on 500 by 500 matrices.
  - 3 for loops: 15 sec.
  - 2 for loops: 2 sec.
  - 1 for loop: 0.2 sec.
  - no for loop: .08 sec.
- Each time you eliminate a `for` loop in your code, your program gets a performance boost.



# Calculating $\pi$ using Monte Carlo

- sample uniformly in the box  $[-1, 1]^2$
- $\pi \approx \text{\#samples in circle} / \text{\#samples} * 4$



# Non-vectorized Code

```
dist <- function(a,b){  
  return(sqrt(sum((a-b)^2)))  
}  
  
n <- 1000000  
#generating n*2 samples from unif(-1,1)  
x <- runif(n*2, -1, 1)  
#create an n by 2 matrix from x  
X <- matrix(x, nrow=n)  
  
count <- 0  
for (i in 1:n){  
  # using a for loop to count samples inside circle  
  if (dist(X[i,],0) < 1){  
    count <- count + 1  
  }  
}  
  
print(count/n * 4) # print pi estimation
```

# Vectorized Code

```
n <- 1000000
#generating n*2 samples from unif(-1,1)
x <- runif(n*2, -1, 1)
#create an n by 2 matrix from x
X <- matrix(x, nrow=n)

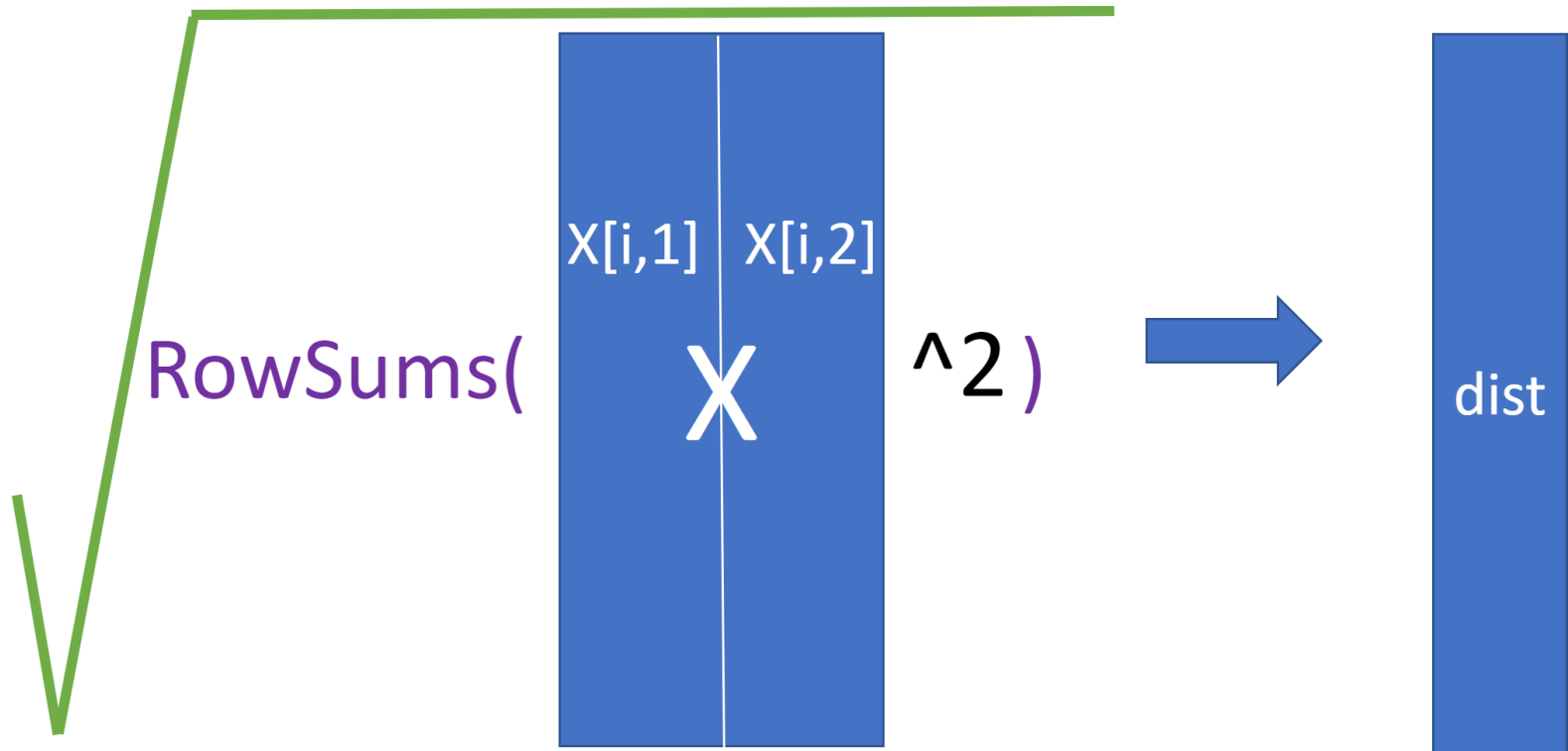
dist <- sqrt(rowSums((X - 0)^2))
count <- length(dist[dist < 1])

print(count/n * 4) # print pi estimation
```

- `rowSums` sums over rows of a matrix.

# Vectorized Code

`dist <- sqrt(rowSums((X - 0)^2))` does the following



- The vectorized code is about 50 times faster than its non-vectorized counterpart.

# Conclusion

- Using vector/matrix operators instead of scalar operators will significantly increase the efficiency of your R code.
- Do not use `for` loop in R unless you absolutely have to.
- In exams, pay attention to the question and see whether it permits vectorized code or not.

# Homework 1: Generating Dummy Data

1. Create **two 500 by 784 matrices** filled with random observations from a **standard** normal distribution.
  - Hint: `rnorm(n,mean,std)` produces a **vector** of size `n` filled with samples from a normal distribution with `mean` as mean and `std` as standard deviation.

## Homework 2: Scalar Compute

2. Write a function called `pdist1` takes two matrices  $A$ ,  $B$  as inputs, and outputs a matrix  $D$ , where  $D_{i,j} = \text{dist}(A_{[i,]}, B_{[j,]})$  and  $\text{dist}(a, b)$  is the Euclidean distance between two vectors  $a, b$ .
- This is the Part II of TB1 project.
  - Your code should have **3 for loops**, in other words, you should write non-vectorized code.
  - Test the function on matrices generated in step 1.

# Homework 2 (submit): Vectorization

3. Write another function called `pdist2`, that does exactly the same thing, using only **two for loops**.
  - Hint, use the `dist` function you wrote in the last week's lab to help you.
4. Write another function called `pdist3`, that does exactly the same thing, using **only one for loop**.
  - Hint, to compute the  $i$ -th row in  $D$ , first compute three terms: `sum(A[i,]^2)`, `rowSums(B[i,]^2)` and `A[i,]%*%t(B)`.
  - Combine them to get the  $i$ -th row of  $D$ .



# Homework 3 (Challenge): Vectoriz.

5. Write another function called `pdist4`, that does exactly the same thing **without using any for loop**.
  - Do not use any built-in function other than `sum` and `rowSums`.
  - Express  $D$  by matrix algebra using  $A$  and  $B$ .
  - Hint: `matrix(k, nrow = M, ncol = N)` will create an  $M$  by  $N$  matrix filled with value  $k$ .
  - Hint: `rowSums` will convert your matrix into a vector. To perform matrix algebra, you need to convert it back to a matrix using `matrix` command.
6. Test all above functions using the matrices you generated in the first step. How long does it take them to run?

