3 - Mathematical Operations

Naturally the vector and matrix arrays as defined in section 2 have mathematical operations which can be applied. These operations include vector/matrix addition, subtraction, element-wise products and standard products. There is a strong similarity between how R and the Python NumPy module perform these operations; in most cases the difference in code is a single character or two.

3.1 - Basic Vector Operations

For numeric vectors $x = [x_1 \dots x_n]^T$ and $y = [y_1 \dots y_n]^T \in \mathbb{R}^n$ the following definitions apply.

Element-wise vector product as:

$$x \odot y = [x_1 \cdot y_1 \dots x_n \cdot y_n]^T \in \mathbb{R}^n$$

Standard vector product as:

$$x \cdot y = x^T y = \sum_{i=1}^{n} x_i y_i \in \mathbb{R}$$

Vector addition as:

$$x + y = [x_1 + y_1 \dots x_n + y_n]^T \in \mathbb{R}^n$$

3.1.1 - NumPy Vector Operations

In Python, operations such as element-wise array multiplication, array products, and array addition are performed by using the built-in command keys `*`, `@` and `+`. Note that when performing addition or multiplication of an array with a scalar, Python will apply the operation across all dimensions.

```
X1 = np.array([1,2,3])
X2 = np.array([2,2,2])
X1, X2
```

```
(array([1, 2, 3]), array([2, 2, 2]))
```

```
X3 = X1 * X2 # element-wise multiplication
X3
```

```
array([2, 4, 6])
```

```
X4 = X1 @ X2 # vector product
X4
```

```
12
```

```
X5 = X1 + X2 # vector addition
X5
```

```
array([3, 4, 5])
```

Notice here below how the dimension is ignored when applying the operation with a scalar value.

```
X6 = X1 + 2 # vector/scalar addition
X6
```

```
array([3, 4, 5])
```

```
X7 = X1 * 2 # vector/scalar multiplication
X7
```

```
array([2, 4, 6])
```

3.1.2 - R Vector Operations

Similarly to Python, in R these operations are performed by using the command keys `*`, `%*%` and `+`. Also when performing addition or multiplication of an array with a scalar, R will apply the operation across all dimensions in the same way as NumPy.

```
X1 <- c(1,2,3)
X2 <- c(2,2,2)
print(X1)
print(X2)
```

```
[1] 1 2 3
[1] 2 2 2
```

```
X3 <- X1 * X2 # element-wise multiplication
print(X3)
```

```
[1] 2 4 6
```

```
X4 <- X1 %*% X2 # vector product
print(X4)
```

```
     [,1]
[1,]   12
```

```
X5 <- X1 + X2 # vector addition
print(X5)
```

```
[1] 3 4 5
```

Notice how R has the same behaviour as Python when applying the operation with a scalar value.

```
X6 <- X1 + 2 # vector/scalar addition
print(X6)
```

```
[1] 3 4 5
```

```
X7 <- X1 * 2 # vector/scalar multiplication
print(X7)
```

```
[1] 2 4 6
```

3.2 - Basic Matrix Operations

For matrices $X = [X_1 \dots X_m] = \begin{bmatrix} x_{11} \dots x_{1m} \\ \vdots \quad \vdots \\ x_{n1} \dots x_{nm} \end{bmatrix}$ and $Y = [Y_1 \dots Y_m] = \begin{bmatrix} y_{11} \dots y_{1m} \\ \vdots \quad \vdots \\ y_{n1} \dots y_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m}$ with $x_{ij},\ y_{ij} \in \mathbb{R}$ and $X_j, Y_j \in \mathbb{R}^n$ for $i = 1, \dots, n,\ j = 1, \dots, m$ one defines:

Element-wise matrix product as:

$$X \odot Y = [X_1 \odot Y_1 \dots X_m \odot Y_m] = \begin{bmatrix} x_{11}y_{11} & \dots & x_{1m}y_{1m} \\ \vdots & & \vdots \\ x_{n1}y_{n1} & \dots & x_{nm}y_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

Standard matrix product as (dimensions must agree):

$$X^T Y = \begin{bmatrix} X_1^T Y_1 & \dots & X_1^T Y_m \\ \vdots & & \vdots \\ X_m^T Y_1 & \dots & X_m^T Y_m \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n x_{1i}y_{1i} & \dots & \sum_{i=1}^n x_{1i}y_{mi} \\ \vdots & & \vdots \\ \sum_{i=1}^n x_{mi}y_{1i} & \dots & \sum_{i=1}^n x_{mi}y_{mi} \end{bmatrix} \in \mathbb{R}^{m \times m}$$

Matrix addition as:

$$X + Y = [X_1 + Y_1 \dots X_m + Y_m] = \begin{bmatrix} x_{11} + y_{11} & \dots & x_{1m} + y_{1m} \\ \vdots & & \vdots \\ x_{n1} + y_{n1} & \dots & x_{nm} + y_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

3.2.1 - NumPy Matrix Array Operations

Operations such as element-wise array multiplication, array products, and array addition are performed the same way for matrices as they are for vectors where dimension sizes must agree. It should be noted that the NumPy transpose operator is simply `.T` which can be used on NumPy arrays of any size.

```
X = np.array([[1,3,5],[2,4,6]])
X
```

```
array([[1, 3, 5],
       [2, 4, 6]])
```

```
M1 = X * X # element-wise multiplication
M1
```

```
array([[ 1,  9, 25],
       [ 4, 16, 36]])
```

```
M2 = X.T @ X # matrix product
M2
```

```
array([[ 5, 11, 17],
       [11, 25, 39],
       [17, 39, 61]])
```

```
M3 = X + X # matrix/matrix addition
M3
```

```
array([[ 2,  6, 10],
       [ 4,  8, 12]])
```

Just as was shown with vectors, NumPy will apply the operation across all dimensions when performing addition or multiplication of an array matrix and a scalar $\in \mathbb{R}$.

```
M4 = X + 2 # matrix/scalar addition
M4
```

```
array([[3, 5, 7],
       [4, 6, 8]])
```

```
M5 = X*2 # matrix/scalar multiplication
M5
```

```
array([[ 2,  6, 10],
       [ 4,  8, 12]])
```

3.2.2 - R Matrix Operations

Matrix operations in R are the same as those used on vectors. It should be noted that the transpose operator in R is simply `t()` which can be applied to both vectors and matrices.

```
X <- matrix(1:6, nrow = 2, ncol = 3)
print(X)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
M1 <- X * X # element-wise multiplication
print(M1)
```

```
     [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36
```

```
M2 <- t(X) %*% X # matrix product
print(M2)
```

```
     [,1] [,2] [,3]
[1,]    5   11   17
[2,]   11   25   39
[3,]   17   39   61
```

```
M3 <- X + X # matrix/matrix addition
print(M3)
```

```
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
```

Again, just as with vectors R will apply the operation across all dimensions if applying the operation with a scalar value.

```
M4 <- X + 2 # matrix/scalar addition
print(M4)
```

```
     [,1] [,2] [,3]
[1,]    3    5    7
[2,]    4    6    8
```

```
M5 <- X*2 # matrix/scalar multiplication
print(M5)
```

```
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
```

3.3 - Basic Matrix-Vector Operations

For vector $x = [x_1 \dots x_n]^T \in \mathbb{R}^n$ and matrix $Y = [Y_1 \dots Y_m] = \begin{bmatrix} y_{11} \cdots y_{1m} \\ \vdots \quad \vdots \\ y_{n1} \cdots y_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m}$ with $Y_j \in \mathbb{R}^n$ and $x_i,\ y_{ij} \in \mathbb{R}$ for $j = 1, \dots, m, i = 1, \dots, n$ one defines:

Standard matrix-vector product (dimensions must agree) as:

$$Y^T x = [Y_1^T x \dots Y_m^T x]^T = [\sum_{i=1}^{n} y_{1i} x_i \dots \sum_{i=1}^{n} y_{mi} x_i]^T \in \mathbb{R}^m$$

3.3.1 - NumPy Matrix-Vector Array Operations Dimensions must agree when computing matrix/vector products. For example, one cannot multiply a matrix of size $\mathbb{R}^{2\times 4}$ with a vector of size $\mathbb{R}^3$. Both Python and R will produce error messages if trying to calculate a matrix/vector product with incompatible dimensions.

```
x = np.array([1,2])
Y = np.array([[1,3,5],[2,4,6]])
x, Y
```

```
(array([1, 2]),
 array([[1, 3, 5],
        [2, 4, 6]]))
```

```
M = Y.T @ x # matrix/vector product
M
```

```
array([ 5, 11, 17])
```

```
M1 = Y.T * x # multiplied across rows
M1
```

```
array([[ 1,  4],
       [ 3,  8],
       [ 5, 12]])
```

Notice above how here each column of Y^T gets multiplied by the value corresponding to the elements of x.

```
x2 = np.array([1,2,3])
M2 = Y * x2 # multiplied across columns
M2
```

```
array([[ 1,  6, 15],
       [ 2,  8, 18]])
```

Above, each column of Y gets multiplied by the value corresponding to the elements of $x2$. Python does not use the same strategy as R in this case. To perform matrix-vector operations, the size of the matrix columns and the vector must match.

```
M3 = X.T + x # addition is the same across rows of X
M3
```

```
array([[2, 4],
       [4, 6],
       [6, 8]])
```

```
M4 = X + x2 # addition is the same across columns of X
M4
```

```
array([[2, 5, 8],
       [3, 6, 9]])
```

3.3.2 - R Matrix-Vector Array Operations

Dimensions must agree to compute matrix/vector products in R. We look to reproduce the operations performed in Python the same way in R.

```
x <- c(1,2)
Y <- matrix(1:6, nrow = 2, ncol = 3)
print(x)
print(Y)
```

```
[1] 1 2
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
M <- t(Y) %*% x # matrix/vector product
print(M)
```

```
     [,1]
[1,]    5
[2,]   11
[3,]   17
```

If the vector in question is of the same size as the number of matrix rows one can perform a version of element-wise matrix/vector multiplication where each row in the matrix is multiplied by the same element from the vector. One should use caution when attempting operations as this since R will perform the matrix/vector multiplication as long as matrix length (number of elements in the matrix) is a multiple of the vector length.

```
M1 <- Y * x # multiply across rows
print(M1)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    4    8   12
```

```
x2 <- c(1,2,3) # multiplication is no longer across rows
M2 <- Y * x2
print(M2)
```

```
     [,1] [,2] [,3]
[1,]    1    9   10
[2,]    4    4   18
```

This "rule" is applied in the same way when performing matrix/vector addition. R will perform the addition as long as matrix length is a multiple of the vector length. If the vector is the same length as the matrix columns, the addition will be applied across the rows.

```
M3 <- Y + x # addition is the same across rows
print(M3)
```

```
     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    4    6    8
```

```
M4 <- Y + x2 # addition is no longer the same across rows
print(M4)
```

```
     [,1] [,2] [,3]
[1,]    2    6    7
[2,]    4    5    9
```

This feature is useful but can lead to unnoticed errors if one is performing such operations with with incorrectly sized objects.

3.4 - Speeding up Matrix Multiplication

If performing matrix/matrix or matrix/vector multiplication, one can use operators such as `@` and `%*%` or built-in NumPy and R functions instead. These choices will all produce the same result but vary in how long they take, thus certain methods should be used to speed up the computation. This section will specifically show the case of computing the matrix multiplication of $X^T X$.

The Python `time` module is used for these examples. Extra documentation on the `time()` function base can be found in the Python standard Library [7]. It is used here only to track the seconds between operations. Using the `time.time()` function, one can extract the current system time at any point during code execution.

```
import time  # for timing code blocks
```

3.4.1 - Python Fast Matrix Multiplication

Consider the matrix multiplication of $X^T X$ (cross product) for matrix X . If making use of the `@` operator, one can use the built-in NumPy `matmul()` function or the `dot()` function to complete identical computations.

```
X = np.array([[1,3,5],[2,4,6]])
M = X.T @ X                      # direct multiplication with @
M_new = np.matmul(X.T, X)        # matmul() function
M_new2 = np.dot(X.T, X)          # dot() function
M, (M == M_new2) == (M == M_new) # all produce same result
```

```
(array([[ 5, 11, 17],
        [11, 25, 39],
        [17, 39, 61]]),
 array([[ True,  True,  True],
        [ True,  True,  True],
        [ True,  True,  True]]))
```

To convince readers of the difference in speed for these functions a simple experiment is run for varying sizes of matrix when computing $X^T X$. One can observe how `matmul()` and `@` perform the same but both perform the matrix multiplication faster than using the `dot()` function.

```
N = np.array([2, 10, 50, 100, 500, 1000, 5000, 7500, 10000, 15000, 20000])

time_original = []
time_function1 = []
time_function2 = []
D = np.random.uniform(0,1, size = (20000,20000))

for i in range(len(N)):
    X = D[0:(N[i]-1), 0:(N[i]-1)]

    t1 = time.time()
    M = X.T @ X
    time_original.append(time.time()-t1)

    t2 = time.time()
    M_new = np.matmul(X.T, X)
    time_function1.append(time.time()-t2)

    t3 = time.time()
    M_new2 = np.dot(X.T, X)
    time_function2.append(time.time()-t3)
```

	Computational Runtime (seconds)		
N for NxN sized matrix	**Original @ operation**	**NumPuy matmul function**	**NumPy dot function**
50	0.000029	0.000015	0.000038
100	0.000097	0.000070	0.001514
500	0.003549	0.003270	0.005038
1000	0.009449	0.008407	0.015854
5000	0.282705	0.222986	1.096729
7500	0.642238	0.625548	1.836163
10000	1.264970	1.227064	3.886481
15000	3.829852	3.969029	9.502815
20000	8.174821	8.056721	19.351828

It should be noted that these computational run times will be different depending on the machine one uses. The main take-away is that these trends will still hold; the `dot()` function is not optimal for computing $X^T X$ whereas there seems to be no clear advantage in using the `@` operator over the `matmul()` function.

3.4.2 - R Fast Matrix Multiplication

Similarly in R, if making use of the `%*%` operator, one can use the built-in `crossprod()` function instead to complete the computation of $X^T X$.

```
X <- matrix(1:6, nrow = 2, ncol = 3)
M <- t(X) %*% X       # direct multiplication with %*%
M_new <- crossprod(X) # can use tcrossprod() for X %*% t(Y)
print(M)
print(M == M_new)     # produce same result
```

```
     [,1] [,2] [,3]
[1,]    5   11   17
[2,]   11   25   39
[3,]   17   39   61
     [,1] [,2] [,3]
[1,] TRUE TRUE TRUE
[2,] TRUE TRUE TRUE
[3,] TRUE TRUE TRUE
```

Similarly to what is done above in Python, one can run a simple experiment in R to test the speed difference in these choices. It is shown how the `crossprod()` function outperforms the direct multiplication `%*%`.

```r
scales <- c(2, 50, 100, 500, 1000, 5000, 7500, 10000, 15000, 20000)
n <- length(scales)
time_original <- numeric(n)
time_function <- numeric(n)
D <- matrix(runif(20000^2, 0, 1), nrow = 20000, ncol = 20000)
count <- 1

for (i in scales) {
    X <- D[1:i,1:i]

    t1 <- Sys.time()
    M_original <- t(X) %*% X
    time_original[count] <- Sys.time() - t1

    t3 <- Sys.time()
    M_function <- crossprod(X)
    time_function[count] <- Sys.time() - t3

    count <- count + 1
}
```

A matrix: 9 x 3 of type dbl

N for NxN sized matrix	Original %*% operation	R crossprod() Function
50	0.000044	0.000024
100	0.008224	0.000532
500	0.030992	0.005052
1000	0.063106	0.008369
5000	1.028639	0.530416
7500	2.777560	1.239570
10000	5.119740	2.580541
15000	14.600272	7.541085
20000	31.476654	16.136891

Again, these computational runtimes will be different depending on the machine one uses. The main take-away is that the R `crossprod()` function is significantly faster than the operation `t(X) %*% X`, with larger matrices causing an even bigger discrepancy between run times. Without getting into the details of why this is the case, R's `crossprod()` function is faster because it does not compute the transpose. Technically the call `t(X) %*% X` has to first take the matrix transpose `t(X)` and then perform the matrix multiplication to get the result, hence performing two separate computations. The `crossprod()` function leverages the structure of X to compute X^TX without having to take the transpose of X at all.

3.5 - Computing Matrix Properties

Computing matrix properties is a very common task in data science. These properties include matrix inverse, pseudo-inverse and matrix determinant. One can also take advantage of matrix structure (PSD matrix) to compute these properties faster and more efficiently.

3.5.1 - Matrix Inverse

For matrix $A \in \mathbb{R}^{n \times n}$ its inverse A^{-1} is defined such that $AA^{-1} = I_n$ for size n identify matrix I_n.

If the matrix A is positive semi-definite (psd), $x^TAx \geq 0 \; \forall x \in \mathbb{R}^n$, then the computation can be done in a more efficient way using a Cholesky decomposition where $A = L^TL$ for triangular matrix L. Thus one can calculate $A^{-1} = L^{-1}L^{-T}$ where L^{-1} is found using a backwards/forwards solver. This is much faster than a normal inverse calculation.

3.5.1.1 - NumPy Matrix Inverse

NumPy provides a linear algebra function base which can be accessed using the call `np.linalg()`. This function base includes a matrix inverse function `inv()`. The inverse can also be computed using a NumPy matrix class `np.matrix()`. For an invertible matrix A the inverse can be computed directly with the call `A.I`.

```r
# 2d NumPy array
M = np.random.uniform(0,1, size = (2,2))
M
```

```
array([[0.0882257 , 0.72253488],
       [0.98528755, 0.27361679]])
```

```r
# Inverse with NumPy function
M_inv1 = np.linalg.inv(M)
M_inv1
```

```
array([[-0.39783495,  1.05055551],
       [ 1.43259419, -0.12827892]])
```

```r
# Inverse with NumPy matrix class
M2 = np.matrix(M)
M_inv2 = M2.I
M_inv1 == M_inv2  # same result
```

```
matrix([[ True,  True],
        [ True,  True]])
```

3.5.1.2 - Python SciPy Module

SciPy is a collection of mathematical algorithms and convenient functions built on the NumPy extension of Python. This section will make use of the SciPy linear algebra module `scipy.linalg` for its matrix inverse and solving functions. As per the SciPy documentation page [9] SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python " `scipy.linalg` contains all the functions in `numpy.linalg` plus some other more advanced ones. Another advantage of using `scipy.linalg` over `numpy.linalg` is that it is always compiled with BLAS/LAPACK support, while for NumPy this is optional. Therefore, the SciPy version might be faster depending on how NumPy was installed."

Once installed it can be loaded/imported with the `import` call as done below. It should be noted that version 1.5.2 of SciPy is being used.

```
import scipy.linalg as la # linear algebra tools
print(scipy.__version__)
```

```
1.5.2
```

Here the linalg functions are imported from SciPy since we are interested in using its `solve_triangular()` function. This function is used to compute the inverse of a triangular matrix efficiently by forward/backward solving.

3.5.1.3 - NumPy/SciPy PSD Matrix Inverse

When computing matrix inverse of a PSD matrix, one can take advantage of this structure to speed up computation. One must compute the Cholesky decomposition of the matrix to finds its inverse. The Cholesky decomposition can be found by using the NumPy function `cholesky()` and the inverse is found with the backwards solver `solve_triangular()`.

```
# Create PSD matrix
M = M + M.T - np.diag(M.diagonal())
M = np.matmul(M,M.T)
M
```

```
array([[10.07986811,  7.31169132],
       [ 7.31169132, 10.47672508]])
```

```
# Normal NumPy inversion
inv1 = np.linalg.inv(M)

# Cholesky/triangular inversion
L = la.solve_triangular(np.linalg.cholesky(M), np.eye(len(M)), lower=True)
inv2 = np.matmul(L.T,L)

# Equivelent results
inv1, inv2
```

```
(array([[ 0.20092226, -0.14022336],
        [-0.14022336,  0.19331135]]),
 array([[ 0.20092226, -0.14022336],
        [-0.14022336,  0.19331135]]))
```

Alternatively, the SciPy linalg `solve()` function has a parameter `sym_pos` which can be set to `True` if the matrix in question is PSD. One can see that this approach also produces the same inverse results.

```
# SciPy psd inversion
inv3 = la.solve(M, np.eye(len(M)), sym_pos = True)
inv3
```

```
array([[ 0.20092226, -0.14022336],
       [-0.14022336,  0.19331135]])
```

When working with PSD matrices, always take advantage of this structure. A small experiment can be run to show how much faster PSD matrix inverse calculations are when using a Cholesky method or built-in PSD method compared to the regular inverse calculation. The internal NumPy `inv()` function is compared to the Cholesky method and to the SciPy PSD inverse method. (SciPy `solve()` has additional parameters to assume the matrix is PSD `assume_a='pos'`.)

```
N = np.array([2, 50, 100, 500, 1000, 5000, 7500, 10000, 15000, 20000])
n = len(N)

X = np.random.uniform(0,1, size = (20000, 20000)) # symmetric psd
X = X + X.T - np.diag(X.diagonal())
X = np.matmul(X,X.T)

time_normal = []
time_chole = []
time_sci = []

for i in range(n):

    M = X[0:N[i], 0:N[i]]
    I = np.eye(N[i])

    t1 = time.time()
    X_inv1 = np.linalg.inv(M)
    time_normal.append(time.time()-t1)

    t2 = time.time()
    L = la.solve_triangular(np.linalg.cholesky(M), I, lower=True, check_finite=False)
    X_inv2 = np.matmul(L.T,L)
    time_chole.append(time.time()-t2)

    t3 = time.time()
    X_inv3 = la.solve(M, I, sym_pos = True, assume_a='pos', overwrite_a=True, overwrite_b=True, check_finite=False)
    time_sci.append(time.time()-t3)
```

Computational Runtime (seconds)

N for NxN Matrix	NumPy Basic Inverse	Cholesky/Triangular Inverse	SciPy Inverse
50	0.000432	0.000211	0.000112
100	0.000458	0.000471	0.000200
500	0.020880	0.005580	0.004286
1000	0.024315	0.048687	0.028815
5000	1.129247	0.899440	0.822636
7500	2.702918	2.209273	2.119805
10000	7.565928	4.326654	4.271921
15000	12.245309	11.531817	13.766104
20000	24.037661	24.183464	26.564049

Note: Run times will vary depending on the machine one uses. The main take-away is that the Cholesky method with NumPy is advantageous for matrices smaller than 20000x20000. One can see that for the large matrix, the NumPy `inv()` function actually out-performs the Cholesky and SciPy methods. This is because the NumPy `inv()` function contains internal multi-processing tools which start to have an effect in decreasing run time if using a large enough matrix. For smaller matrices the multi-processing tools do not have as much of an effect so the Cholesky and SciPy methods are faster.

3.5.1.4 - R Matrix Inverse

R has a built in `solve()` function which can be used to compute the matrix inverse. `solve()` is used to find solutions for $Ax = b \Rightarrow x = A^{-1}b$ but calling `solve(A)` will compute A^{-1} directly.

```
M <- matrix(runif(4, 0, 1), ncol = 2)
print(M)
```

```
          [,1]      [,2]
[1,] 0.3010248 0.4155825
[2,] 0.1056157 0.5757500
```

```
M_inv <- solve(M) # to find inverse.
print(M_inv)
```

```
           [,1]      [,2]
[1,]  4.4485912 -3.211041
[2,] -0.8160505  2.325899
```

```
print(M %*% M_inv) # check for idenity
```

```
             [,1]          [,2]
[1,]  1.00000e+00 -5.321529e-17
[2,] -3.12986e-18  1.000000e+00
```

3.5.1.5 - R PSD Matrix Inverse

In R the PSD matrix inverse can also use the same Cholesky strategy to speed things up. This is done with the R functions `chol()` and `backsolve()`.

```
M <- M + t(M) - diag(diag(M)) # symmetric psd M
M <- crossprod(M)

L <- chol(M)                          # cholesky
L_inv <- backsolve(L, diag(ncol(L))) # triangular inverse
inv1 <- tcrossprod(L_inv)
inv2 <- solve(M)
print(inv1)
print(inv2)                           # same results
```

```
          [,1]      [,2]
[1,]  62.37641 -47.26029
[2,] -47.26029  37.46537
          [,1]      [,2]
[1,]  62.37641 -47.26029
[2,] -47.26029  37.46537
```

A small experiment can also be run in R to show how much faster PSD matrix inverse calculations are if using a Cholesky decomposition compared to the regular inverse function.

```
scales <- c(2, 50, 100, 500, 1000, 5000, 7500, 10000, 15000, 20000)
n <- length(scales)
time_solve <- numeric(n)
time_chol <- numeric(n)

# Generate psd matrix
X <- matrix(runif(20000^2, 0, 1), nrow = 20000, ncol = 20000)
X <- 0.5 * (t(X) + X) - diag(diag(X))
X <- crossprod(X)
count <- 1

for (i in scales) {
    M <- X[1:i, 1:i]

    t1 <- Sys.time()
    inv1 <- solve(M)
    time_solve[count] <- Sys.time() - t1

    t3 <- Sys.time()
    L <- chol(M)
    L_inv <- backsolve(L, diag(ncol(L)))
    inv2 <- tcrossprod(L_inv)
    time_chol[count] <- Sys.time() - t3
```

```
    count <- count + 1
}
```

A matrix: 9 x 3 of type dbl

N for NxN sized matrix	Original 'solve()' Inverse	Cholesky Inverse Method
50	0.001137	0.000117
100	0.005223	0.000315
500	0.456043	0.005908
1000	1.421121	0.026170
5000	5.334972	0.847194
7500	7.199269	1.962065
10000	10.100626	4.224895
15000	20.104878	10.673056
20000	35.091830	24.792031

Note: Run times will vary depending on the machine one uses. The main take-away is that use of the Cholesky method in R will significantly reduce computational run time. This difference in speed will increase as the size of the matrix in question increases. This should be great motivation to always take advantage of PSD matrix structure in R.

3.5.2 - Matrix Pesudo-Inverse

If the matrix inverse does not exist one can calculate its pseudo inverse. The pseudo inverse of a matrix $A \in \mathbb{R}^{n \times m}$ notation A^+, is defined as:

$A^+ = (A^T A)^{-1} A^T$ if A is full column rank,
$A^+ = A^T (A A^T)^{-1}$ if A is full row rank and
$A^+ = A^{-1}$ if A is square, invertible

3.5.2.1 - SciPy Pseudo Inverse

This is done with the SciPy linear algebra function base. The `pinv()` function computes the matrix pseudo inverse.

```
A = np.array([[1,2],[3,4], [5,6]])
A
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
A_pinv = la.pinv(A)
A_pinv @ A # approximates inverse so that A*A^{-1} = I
```

```
array([[1.0000000e+00, 8.8817842e-16],
       [0.0000000e+00, 1.0000000e+00]])
```

3.5.2.2 - R Package pracma (Practical Numerical Math Functions)

As per the pracma documentation page [10] Package 'pracma' (Practical Numerical Math Functions), the package 'Provides a large number of functions from numerical analysis and linear algebra, numerical optimization, differential equations, time series, plus some well-known special mathematical functions. Uses 'MATLAB' function names where appropriate to simplify porting.' One can load this package with the call `library(pracma)` if already installed. If not installed, one can download the package from CRAN using the call `install.packages('pracma')`. We will use the peudo-inverse function in pracma. It should be noted that version 2.2.9 of the pracma package is being used.

```
# install.packages('pracma') -> run if not installed
library(pracma)
packageVersion("pracma")
```

```
[1] '2.2.9'
```

3.5.2.3 - R Pseudo Inverse

We will make use of the pracma function `pinv()` to calculate matrix pseudo inverse.

```
A <- matrix(c(1,3,5,2,4,6), ncol = 2)
print(A)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

```
A_pinv <- pinv(A)
print(A_pinv %*% A) # approximate inverse so that A*A^{-1} = I
```

```
              [,1] [,2]
[1,]  1.000000e+00    0
[2,] -4.440892e-16    1
```

3.5.3 - Matrix Determinant

For matrix $A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \in \mathbb{R}^{n \times n}$ let $A(i,j)$ be the $\mathbb{R}^{(n-1)\times(n-1)}$ sized matrix formed by deleting the ith row and jth column of A.

The determinant of A can then be defined as:

$$det(A) = |A| = \sum_{i=1}^{n} a_{i1} \cdot (-1)^{i+1} \cdot det(A(i,1))$$

Again, if the matrix A is positive semi-definite (PSD), $x^T A x \geq 0 \; \forall x \in \mathbb{R}^n$, then the computation can be done in a more efficient way using a Cholesky decomposition where $A = L^T L$ for triangular matrix L. Since L is triangular one can calculate the determinant of the PSD matrix as:

$$det(A) = det(L^T) \cdot det(L) = (\prod_{i=1}^{n} L_{ii})^2$$

3.5.3.1 - SciPy Matrix Determinant

To calculate determinant in Python it is recommended to use the SciPy linear algebra function base instead of NumPy. They are virtually the same but SciPy has more options and tends to be more powerful in what it can accomplish. The determinant can be called by using `.det()`.

```
X = np.array([[1,2],[3,4]])
det = la.det(X)
X, det
```

```
(array([[1, 2],
        [3, 4]]),
 -2.0)
```

3.5.3.2 - SciPy PSD Matrix Determinant

For a PSD matrix, one can again take advantage of this structure to speed up computation. One must compute the Cholesky decomposition of the matrix to find the determinant. The determinant of a triangular matrix is the product of its diagonal values.

```
X = 4 * np.eye(4)
det_eye = la.det(X)
X, det_eye
```

```
(array([[4., 0., 0., 0.],
        [0., 4., 0., 0.],
        [0., 0., 4., 0.],
        [0., 0., 0., 4.]]),
 256.0)
```

```
L = la.cholesky(X)
det2 = L.diagonal().prod() ** 2
det2 # same result
```

```
256.0
```

A small experiment is run comparing normal vs Cholseky methods to convince readers of the speed increase.

```
N = np.array([2, 50, 100, 500, 1000, 5000, 7500, 10000, 15000, 20000])
n = len(N)

X = np.random.uniform(0,1, size = (20000, 20000)) # symmetric psd
X = X + X.T - np.diag(X.diagonal())
X = np.matmul(X,X.T)/20000

time_normal = []
time_chole = []

for i in range(n):

    M = X[0:N[i], 0:N[i]]

    t1 = time.time()
    X_inv1 = la.det(M)
    time_normal.append(time.time()-t1)

    t2 = time.time()
    L = la.cholesky(M)
    X_inv2 = L.diagonal().prod() ** 2
    time_chole.append(time.time()-t2)
```

Computational Runtime (seconds)

N for NxN Matrix	SciPy Basic Determinant	Cholesky Determinant
50	0.000381	0.000075
100	0.000442	0.000169
500	0.904340	0.003077
1000	2.433227	0.043376
5000	8.525671	0.601734
7500	10.126296	1.279135
10000	12.639214	1.540401
15000	16.681865	3.881504

20000	21.266501	6.762789

Note: Run times will vary depending on the machine one uses. The main take-away is that using a Cholesky method to find the determinant of a PSD matrix is significantly more efficient than using a normal determinant calculation. The difference in speed will continue to increase as the size of the matrix in question increases. This should be sufficient motivation to use a Cholesky method to calculate the determinant of a PSD matrix in Python.

3.5.3.3 - R Matrix Determinant

In R one can utilize the built in function `det()` to compute matrix determinant.

```
X <- matrix(c(1,2,3,4), ncol = 2)
print(X)
det(X)
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
-2
```

3.5.3.4 - R PSD Matrix Determinant

For PSD matrices the computation can again be greatly sped up. R uses the `diag()` function to extract diagonal elements in a matrix and the `prod()` function to compute the product of an array of elements.

```
X <- matrix(runif(4, 0, 1), nrow = 2, ncol = 2)
X <- 0.5 * (t(X) + X)
X <- crossprod(X)
det1 <- det(X)
print(X)
```

```
         [,1]     [,2]
[1,] 1.452562 1.448369
[2,] 1.448369 1.481538
```

```
L <- chol(X)
det2 <- prod(diag(L))**2
# same results
det1
det2
```

```
0.0542528720289448
0.0542528720289448
```

A small experiment can be run to convince readers of the speed increase.

```
scales <- c(2, 50, 100, 500, 1000, 5000, 7500, 10000, 15000, 20000)
n <- length(scales)
time_solve <- numeric(n)
time_chol <- numeric(n)
X <- matrix(runif(20000^2, 0, 1), nrow = 20000, ncol = 20000)
X <- 0.5 * (t(X) + X) - diag(diag(X))
X <- crossprod(X)

for (i in 1:n) {
  N <- scales[i]
  M <- X[1:N, 1:N]

  t1 <- Sys.time()
  det1 <- det(M)
  time_solve[i] <- Sys.time() - t1

  t3 <- Sys.time()
  L <- chol(M)
  det2 <- prod(diag(L)) ** 2
  time_chol[i] <- Sys.time() - t3
}
```

A matrix: 9 x 3 of type dbl

N for NxN sized matrix	Original 'det()' Determinant	Cholesky Determinant Method
50	0.001460	0.000672
100	0.005022	0.000138
500	0.413112	0.002762
1000	1.334488	0.030634
5000	4.695797	0.261089
7500	6.188897	0.708372
10000	7.033950	1.471535
15000	10.149708	3.090780
20000	14.394160	6.171631

Note: Run times will vary depending on the machine one uses. The main take-away is the same as the Python experiment. Using a Cholesky method to find the determinant of a PSD matrix is clearly superior in R.

There are other computational methods besides a Cholesky decomposition such as QR decomposition or singular value decomposition (SVD) which can be used to one's advantage. Only the Cholesky method was explored here because of its general speed advantage and simplicity to implement. It should be noted that a Cholesky decomposition is not always optimal and that one should always be aware of the structure and properties of the mathematical objects one is dealing with.

[7] Van Rossum, G. & Drake, F.L., 2009. Python 3 Reference Manual, [link]
[9] Virtanen, P. et al., 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, [link]
[10] Hans W. Borchers, 2022, Package 'pracma' (Practical Numerical Math Functions), [link]