PHY 407 Final Project

The Cholesky Decomposition and its uses.

Introduction

The Cholesky Decomposition is a decomposition of positive definite (i.e. All eigenvalues are positive) Hermitian matrix into a lower-triangular matrix, and by extension, its conjugate transpose. When applicable, it is a useful tool in solving numerous problems in linear algebra and physics in various ways, some of which are discussed in this report.

Computational Background - Decompositions

The Cholesky Decomposition takes a square, positive definite matrix **A** (Any matrix can be "converted" to a positive-definite matrix by taking $\mathbf{A}^{*}\mathbf{A}$) and produces a lower-triangular matrix **L**. This matrix **L** is different depending on whether the classic decomposition (producing **L** and **L***) or the LDL decomposition, where it produces **L** and **L*** matrices, lower- and upper-triangular respectively, with all diagonal elements equalling 1, and producing an additional **D** matrix where the diagonal elements are non-zero and the other elements are zero.

Classic Decomposition: To produce matrix **L**, one can use the formula

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk} L *_{jk}}$$

to first calculate the diagonal elements of the matrix, so that the following forward substitution

$$L_{ij} = \frac{A_{ij} - \sum_{k=1}^{j-1} L_{ik} L *_{jk}}{L_{jj}}$$

can then be used to calculate the rest of the elements, for which i > j. This produces **L** matrix, and is the philosophy used behind the code for this project.

LDL Decomposition: To produce matrix **L** here, one must first calculate

$$\mathrm{D_{jj}} = \mathrm{A_{jj}} - \sum_{\mathrm{k=1}}^{\mathrm{j-1}} \mathrm{L_{jk} L *_{jk} D_{kk}}$$

to obtain the diagonal elements of the **D** matrix. Following this, one can then use

$$L_{ij} = \frac{A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{*jk} D_{kk}}{D_{jj}}$$

to calculate the elements of the **L** matrix. Notably, this method doesn't involve using a square root to calculate the elements, so this decomposition can also be used on positive semi-definite matrices (i.e. All eigenvalues are greater or equal to zero, rather than just being greater than 0).

Compared to the numpy.linalg.cholesky function, these functions are much less efficient, taking a noticeable amount of time for large matrices compared to the numpy function. The time taken as n increases will also be observed later.

Computational Background – Applications

There are 3 (and an extra) applications for this decomposition. Ideally, I would've performed more analysis on the decomposition, such as more precise floating-point error analysis, and comparing LU decomposition to Cholesky decomposition for matrices that are well-defined, but unfortunately this was all I could fit in. The first is **matrix inversion**, which uses the decomposed **L** and L* matrices to calculate the inverse of a matrix without explicit equation solving. This is efficient for large matrices that naturally are too tedious to be done manually or computational expensive to be done by other methods, such as Triangular operations.

This is done by first taking the relation $\mathbf{AA^{-1}=I_{nxn}}$, where A and I are known. By decomposing A, we get $\mathbf{LL^*A^{-1}=I_{nxn}}$. Taking $\mathbf{L^*A^{-1}=u}$, this finally becomes $\mathbf{Lu = I_{nxn}}$, which can be solved via forward substitution. Solving **u**, the earlier relation $\mathbf{L^*A^{-1} = u}$ can now be taken and solved via backward substitution to obtain the inverse matrix of **A**, $\mathbf{A^{-1}}$.

Translated into Python, this worked well for most matrix sizes, in that the Cholesky method using my manual function give values very close or the same as the numpy.linalg.inv function. However, due to floating point errors and other minor discrepancies for various elements in the matrices, which were at times more evident as the size of the matrices increased. Regardless, this method when applicable is very fast compared to other methods of inversion, **requiring $\boldsymbol{O(\frac{1}{2}n^3)}$ operations** compared to triangular matrix operations, which require $O(\frac{2}{3}n^3)$ operations, and regular equation solving, which require $O(\frac{5}{6}n^3)$ operations.

The second application was **Monte Carlo Correlation**. This was simply taking **n** data sets of randomly generated values representing **n** uncorrelated variables, and using a matrix to generate a positive definite matrix via $\mathbf{A^*A}$. This positive definite matrix was then used to create a lower-triangular matrix **L**

that then correlated the variables by dotting the i^{th} row of the matrix with every i^{th} data set. This produced a correlated dataset as can be seen in the graphs below.

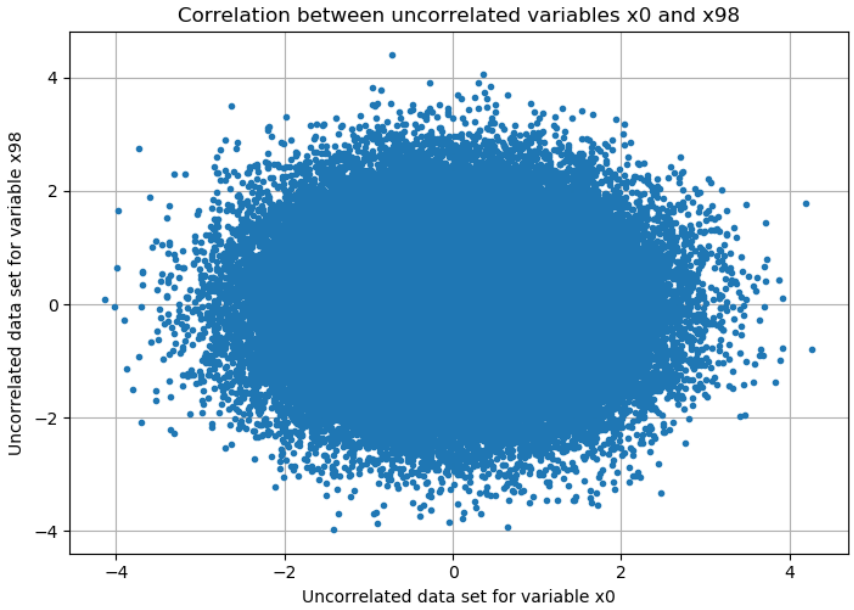

Figure 1: For matrix of size 100 and 100 variables, this is what the correlation between the 1st variable and 99th variable looks like when they are uncorrelated.

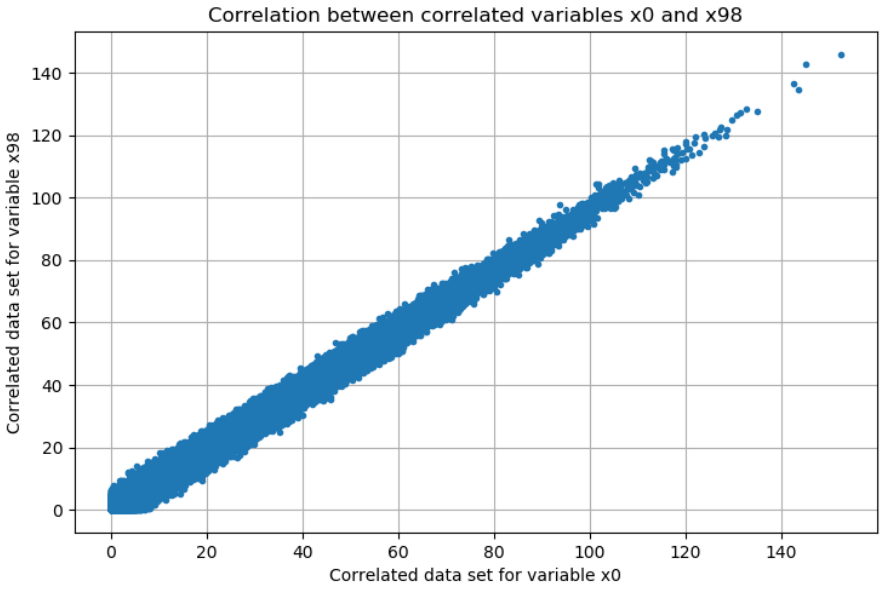

Figure 2: For matrix of size 100 and 100 variables, this is what the correlation between the 1st variable and 99th variable looks like when they are correlated with the randomly generated matrix.

This correlation gives a linear plot between the two variables and can be used to perform large simulations/approximations between two variables, such as the correlation between the intensity of

light and the directional cosine of photons exiting a star, which was discussed in Lab 10. While I did attempt to use this on the data from that Lab, I was unable to make it sensibly work, but due to the linear nature of the relation between the directional cosine and intensity, this should be possible.

The last application in this report is **Least Squares Linear Regression**. Following from the methods used to calculate the inverse of a matrix and to find the Cholesky decomposition itself, a matrix **A** (doesn't have to be positive definite itself) and a corresponding solution vector **b** in **Ax=b** can be solved via least squares linear regression to obtain vector **x**. This was done by taking $\mathbf{A}^*\mathbf{A}$ to obtain a positive definite matrix, and then obtaining **L** and **L*** from this matrix. Using forward and backward substitution, these matrices were then used to obtain the vector **x**.

As will be shown soon, this method is in fact quite unstable for large matrices but works well enough for smaller ones. It should be noted that, for some reason that I couldn't pinpoint out, at rare sizes and seeds the matrix doesn't work for this method. Changing the seed usually fixes this.

Matrix Testing, n = 3

At n = 3, the matrix is small enough that operations don't suffer from stacking floating point error or minor decimal value differences that would otherwise eventually add up to cause visible discrepancies.

```
The matrix inversion from numpy.linalg is
[ 1524.77484582   187.81659733 -2473.80495161]
[  187.81659733    86.41124493  -363.31524773]
[-2473.80495161  -363.31524773  4099.27446127]]
The matrix inversion from manual method is
[ 1524.77484582   187.81659733 -2473.80495161]
[  187.81659733    86.41124493  -363.31524773]
[-2473.80495161  -363.31524773  4099.27446127]]

The linear equation for a solution
[[0.10905042]
 [0.88977662]
 [0.86960725]]
via linalg.solve from cholesky gives vector x =
[[-1817.84656138]
 [ -218.57338841]
 [ 2971.71990065]]
The linear equation for a solution
[[0.10905042]
 [0.88977662]
 [0.86960725]]
via least squares linear regression from cholesky gives vector x =
[-1817.84656148  -218.57338842  2971.71990082]
The average ratio between the numpy and manual method at n = 3  for matrix inversion is 1.0000000000000238
The average ratio between the numpy and manual method at n = 3  for least squares is 0.9999999999421397
Total time taken to run code =  0.07480025291442871
```

Figure 3: As can be seen in the code printouts above, the manual methods give values very similar to the numpy methods, which I take as the more accurate methods. The ratios are very close to 1, which implies that these methods work fine for a matrix of this size.

Matrix Testing, n = 50

There wasn't much of a difference between this size and the previous one, as both methods were still numerically stable.

```
The average ratio between the numpy and manual method at n = 50  for matrix inversion is 0.9999999999998006
The average ratio between the numpy and manual method at n = 50  for least squares is 1.000000024173999
Total time taken to run code =  1.079113245010376
```

Figure 4: The ratios at n=50. The matrices are too large to be shown here.

Matrix Testing, n = 200

This size demonstrates the numerical instability of my method for solving linear equations via least squares.

```
The average ratio between the numpy and manual method at n = 200  for matrix inversion is 1.0000000128119257
The average ratio between the numpy and manual method at n = 200  for least squares is 97.85176550527162
Total time taken to run code =  48.1771285533905
```

Figure 5: The ratios are n = 200.

The average ratio between the numpy method for calculating least squares and my method is 97.85, which is almost a 100 fold difference between the actual values and my values. The matrix inversion method, however, is still quite stable, as the ratio is still very close to 1.

Backtracking to various sizes between 100 and 200, the ratios are:

```
The average ratio between the numpy and manual method at n = 100  for matrix inversion is 0.9999999999747928
The average ratio between the numpy and manual method at n = 100  for least squares is 0.9997736299821753
Total time taken to run code =  6.440770864486694

The average ratio between the numpy and manual method at n = 110  for matrix inversion is 1.0000000000975957
The average ratio between the numpy and manual method at n = 110  for least squares is 1.0009045526554288
Total time taken to run code =  8.728651523590088

The average ratio between the numpy and manual method at n = 120  for matrix inversion is 0.999999999698684
The average ratio between the numpy and manual method at n = 120  for least squares is 0.8834730567869683
Total time taken to run code =  10.800108671188354

The average ratio between the numpy and manual method at n = 130  for matrix inversion is 1.0000000000612073
The average ratio between the numpy and manual method at n = 130  for least squares is 1.0008652783213319
Total time taken to run code =  13.677427768707275

The average ratio between the numpy and manual method at n = 140  for matrix inversion is 1.000000000004186
The average ratio between the numpy and manual method at n = 140  for least squares is 1.0001095773119804
Total time taken to run code =  17.03942060470581

The average ratio between the numpy and manual method at n = 150  for matrix inversion is 0.9999999999924948
The average ratio between the numpy and manual method at n = 150  for least squares is 0.9998541363796789
Total time taken to run code =  20.805861234664917

The average ratio between the numpy and manual method at n = 160  for matrix inversion is 1.0000000000407705
The average ratio between the numpy and manual method at n = 160  for least squares is 1.003147100239805
Total time taken to run code =  24.697946071624756

The average ratio between the numpy and manual method at n = 170  for matrix inversion is 0.999999999933551
The average ratio between the numpy and manual method at n = 170  for least squares is 1.0010864692881174
Total time taken to run code =  29.551971673965454

The average ratio between the numpy and manual method at n = 180  for matrix inversion is 1.0000000000952567
The average ratio between the numpy and manual method at n = 180  for least squares is 0.9981382088469094
Total time taken to run code =  34.75007390975952

The average ratio between the numpy and manual method at n = 190  for matrix inversion is 0.9999999998222222
The average ratio between the numpy and manual method at n = 190  for least squares is 1.014055079555022
Total time taken to run code =  41.51048302650452
```

```
The average ratio between the numpy and manual method at n = 195  for matrix inversion is 0.9999999380003727
The average ratio between the numpy and manual method at n = 195  for least squares is 3412.571479075985
Total time taken to run code =  44.67606234550476
The average ratio between the numpy and manual method at n = 198  for matrix inversion is 0.9999998382208078
The average ratio between the numpy and manual method at n = 198  for least squares is 1728.6691228830246
Total time taken to run code =  46.271745681762695
```

This instability seems to happen exactly at around n=194 or n=195. From here onwards however, it fluctuates between stability and instable phases, as shown at n=300.

```
The average ratio between the numpy and manual method at n = 300  for matrix inversion is 0.9999999991837867
The average ratio between the numpy and manual method at n = 300  for least squares is 0.9447656127656042
Total time taken to run code =  158.02927994728088
```

Figure 6: Ratios at n=300. The ratio between numpy and manual for least squares isn't as close to 1 as it was before, but is still relatively close compared to values at n=200.

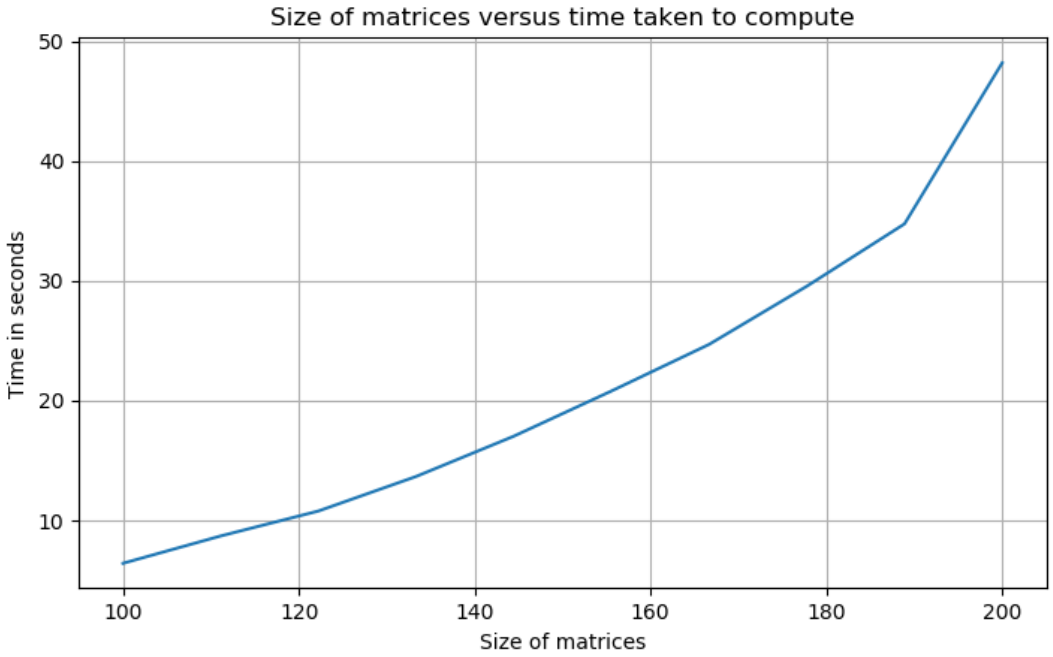

Figure 7: The time taken to compute these processes also increases quite rapidly as n increases, due to the row and column operations that it works by.

The time taken for the code to operate increases rapidly as the size of the matrices increase. This is evident between n = 3 and n = 100, where the time difference is only around 5 seconds long, compared to n = 100 and n = 200, where the time difference is around 42 seconds. Compared to the numpy/Fortran code, this is much slower, and as demonstrated above, more prone to instability.

Extra: Cholesky Cipher

I wished to try something relevant to the decomposition but not necessarily Physics related (in hindsight probably not the best idea), and something that I've never seen before. Since Cholesky decompositions are unique to each matrix, one could theoretically create or randomly generate a matrix with a size

corresponding to the amount of characters they wish to encode, make it positive definite, and decompose it so that the diagonal elements (or other elements/combinations of elements) can be taken and assigned to characters to encode them.

This was achieved in the “choleskyencoder.py” and “choleskydecoder.py” files. They import a few premade paragraphs and encode/decode them respectively. The matrix and seed have to be precisely the same between the encoder and decoder, otherwise it won’t work.

```
In [111]: spliceddata
Out[111]: array(['S', 'o', 'm', ..., 't', 's', '.'], dtype='<U32')

In [112]: encoded
Out[112]:
array([2.40686202, 1.28801424, 1.33512106, ..., 0.57666245, 2.09899029,
       2.56317379])
```

Figure 8: Part of the final report manual encoded using elements from a decomposed 41x41 matrix.

```
n [113]: runfile('C:/Users/user/Desktop/PHY407 Final Project/choleskydecoder.py', wdir='C:/Users/user/Desktop/PHY407 Final Project')
eloaded modules: cipherexamples
'S' 'o' 'm' ... 't' 's' '.']
o m e  p r o j e c t s  m i g h t  i n v o l v e  c o m p l e x  c o m p u t i n g  m e t h o d s  b u t  o n l y  a  f e w  l i n e s  o f  c
d e  t o  i m p l e m e n t  t h e s e  w o u l d  p r o b a b l y  r e s u l t  i n  a  p r o j e c t  w i t h  a  m u c h  l o n g e r  i n
r o d u c t i o n  o r  c o m p u t a t i o n a l  b a c k g r o u n d  e x p l a i n i n g  t h e s e  c o m p l e x  m e t h o d s ,  b u t
h e  c o d e  w o u l d  b e  f a i r l y  s h o r t .  S o m e  m i g h t  i n v o l v e  f a i r l y  s i m p l e  m e t h o d s  t o  u n
e r s t a n d ,  b u t  r e q u i r e  a  l o t  o f  p r o g r a m m i n g  i n  t h i s  c a s e ,  t h e  i n t r o  c o m p u t a t i o n
l  b a c k g r o u n d  m a y  b e  s h o r t e r  b u t  t h e  c o d e s  w i l l  b e  l o n g e r .  B a s e d  o n  t h e  w e i g h t
f  t h e  t e r m  p r o j e c t  a n d  t h e  a m o u n t  o f  t i m e  y o u  h a v e  t o  d o  i t ,  I  a m  e x p e c t i n g  a n  a
o u n t  o f  w o r k  t h a t  i s  t h e  e q u i v a l e n t  o f  2  r e g u l a r  c o u r s e  l a b s  i n  t e r m s  o f  t i m e  s
e n t  o n  t h e  p r o j e c t .  R e p o r t s  w i l l  b e  d u e  W e d n e s d a y  D e c .  4 ,  2 0 1 9 .  H o w e v e r ,  t h e r
 i s  a  N O  P E N A L T Y  e x t e n s i o n  u n t i l  W e d n e s d a y  D e c .  1 8  a t  5  0 0 p m .  T h i s  e x t e n s i o n  d e
d l i n e  i s  s t r i c t .  M a k e  s u r e  t o  h a n d  i n  y o u r  p r o j e c t  l o n g  b e f o r e  t h i s  d e a d l i n e  s
 t h a t  t h e r e  i s  n o  i s s u e  w i t h  b e i n g  j u s t  o v e r  t h e  d u e  d a t e  t i m e .  W o r k  h a n d e d  i n  a
t e r  W e d n e s d a y  D e c .  1 8  a t  5  0 0 p m  w i l l  b e  a s s i g n e d  a  g r a d e  o f  0 .  H o w  t o  h a n d  i n  s i
i l a r  t o  t h e  l a b s ,  y o u  w i l l  h a n d  i n  y o u r  p r o j e c t  v i a  Q u e r c u s .  S a m e  r u l e s  a s  t h e
s s i g n m e n t s .
```

Figure 9: And the same part decoded.

References

- **Web links:**
  - https://arxiv.org/ftp/arxiv/papers/1111/1111.4144.pdf, for matrix inversion
  - https://en.wikipedia.org/wiki/Cholesky_decomposition, for general information
  - https://www.goddardconsulting.ca/option-pricing-monte-carlo-basket.html, understanding the Monte Carlo Correlation
- **Documents**
  - Lab Manuals and Lecture Notes, PHY407, University of Toronto