Assignment 1 - Matrix-matrix multiplication

This assignment makes up 20% of the overall marks for the course. The deadline for submitting this assignment is 5pm on Monday 21st October 2024.

Coursework is to be submitted using the link on Moodle. You should submit a single PDF file containing your code, the output when you run your code, and your answers to any text questions included in the assessment. The easiest ways to create this file are:

Write your code and answers in a Jupyter notebook, then select File -> Download as -> PDF via LaTeX (.pdf).

Write your code and answers on Google Colab, then select File -> Print to print them as a PDF.

The tasks you must carry out and the questions you must answer are shown in bold below.

The assignment

In this assignment, we will look at computing the product AB of two matrices $A,B\in\mathbb{R}n\times n$. The following code snippet defines a function that computes the product of two matrices. For example, the product of two 10 by 10 matrices is printed. The final line prints matrix1 @ matrix2 - the @ symbol denotes matrix multiplication, and Python will get Numpy to compute the product of two matrices. By looking at the output, it's possible to check that the two results are identical.

```python
import numpy as np
import timeit
import matplotlib.pyplot as plt

def slow_matrix_product(mat1, mat2):
    """Multiply two matrices."""
    assert mat1.shape[1] == mat2.shape[0]
    result = []
    for c in range(mat2.shape[1]):
        column = []
        for r in range(mat1.shape[0]):
            value = 0
            for i in range(mat1.shape[1]):
                value += mat1[r, i] * mat2[i, c]
            column.append(value)
        result.append(column)
    return np.array(result).transpose()


matrix1 = np.random.rand(20, 20)
matrix2 = np.random.rand(20, 20)
```

```python
print(slow_matrix_product(matrix1, matrix2))
print(matrix1 @ matrix2)
```

```
[[4.63702448 4.00488453 3.95522105 4.3789378  4.06176472 6.50679293
  4.77610576 5.18026506 5.43375745 3.68733768 5.44798516 5.53971301
  5.22864341 7.36354183 4.69340147 4.84478703 5.47757704 4.12367848
  5.60415434 3.00976476]
 [4.29925627 5.04363664 2.87654607 4.54102703 4.15759764 4.74121209
  4.63572594 5.12392859 4.47593651 4.33363443 5.06181023 6.06476145
  4.77180514 6.45983505 4.15387961 4.69511556 4.70694237 4.22200305
  4.38602656 3.15149349]
 [4.96381087 5.66338601 4.65574156 5.47686973 5.25968199 7.00427169
  5.86144816 6.12315044 6.32488818 5.09630194 7.20214127 7.44667657
  5.11769343 8.84473486 5.94871368 5.86300717 6.13296376 5.79212848
  6.09076094 4.70590872]
 [3.78902405 3.93362481 4.01241918 3.29401633 3.84989007 4.89406138
  3.78610562 4.35577752 4.30408909 3.13783986 5.08438961 5.0340093
  4.09319394 6.70995704 4.02036172 4.27173153 4.63853685 3.70733642
  4.59210347 3.22371253]
 [3.14900379 3.16740219 2.06164627 3.13969426 3.02967164 3.75436913
  3.3702899  3.75404068 3.47030645 3.1183955  3.86426625 4.02971287
  3.55351106 4.71747241 3.65011727 3.50828188 3.77980197 3.00203675
  3.4424578  2.80047338]
 [3.49162826 4.77443846 3.14653553 4.73107315 4.1584574  5.07381921
  4.50977718 5.11257024 4.45667501 3.5379521  5.46228471 5.12060025
  4.60466859 6.69050065 4.01544832 4.85247315 5.30289004 4.21803987
  4.27673867 3.43547668]
 [3.30197086 3.16063463 3.13435868 3.43208713 2.78414018 4.59130127
  3.72678482 3.83201097 4.14642673 2.56235183 3.89392687 3.96436895
  3.68545876 5.22423285 3.69562161 3.24283721 4.06810341 3.39418248
  4.47197951 2.82054583]
 [4.13832891 5.25431609 3.62557884 4.73876228 4.4127073  5.99077609
  5.11242383 5.61753873 5.70581366 4.2126438  5.09883775 6.01067667
  4.80083055 7.65903446 4.57144045 4.54128483 5.94087051 4.54891694
  5.1315085  3.11247694]
 [5.13521299 5.5608853  5.0593564  5.82321474 5.39704587 6.78302327
  6.48769961 6.70449365 6.59059167 4.03895663 6.85172016 7.30673603
  5.97315875 8.86129836 5.83587961 6.39003051 6.67574901 5.40803169
  6.3885298  4.89191892]
 [3.68493497 3.47206051 3.2780746  4.33465906 4.09016041 4.58848072
  3.36772616 4.6017977  3.63108525 3.05957676 4.85599618 5.09139752
  4.14511414 6.20435562 4.25651063 5.14177892 5.19375251 3.41841606
  4.20871593 3.37121056]
 [4.93498254 6.34455419 4.42132026 5.99520938 5.12706902 6.85305969
  6.43472797 6.73489273 6.32455272 5.66886088 5.96587062 7.19850398
  6.20743213 9.21207209 5.2608264  5.8772283  6.43761049 5.31557252
  6.51244106 4.6404681 ]
 [3.33581865 4.18695016 2.87540272 3.80991169 3.93366442 4.76989052
  4.61857207 4.5279083  4.63445457 3.04085352 5.24580175 4.84231847
  4.46864152 6.10288021 3.49837375 4.10939816 5.11703362 3.78477507
  3.66764909 2.93006523]
 [3.30872372 3.92886087 3.45767273 4.15103667 3.81246504 4.13924995
  4.02071017 4.43012935 3.9240151  2.85147715 4.87761493 5.58511617
  3.79142515 5.847782   3.71523084 4.22465862 3.96365732 3.44447769
  4.37304689 2.98120635]
 [3.84089899 4.06287974 2.70315815 4.37998991 4.18930731 5.3761865
  5.07630943 5.38302457 4.96539644 4.44692224 5.25237338 5.11202151
  4.82402705 6.47816027 4.94809273 4.91040691 5.02022561 3.6581127
  4.36661232 3.77809113]
 [3.9659709  4.40100824 3.09104657 5.13349265 4.89710378 5.21532118
  4.69109561 5.50864695 4.86928118 4.32130978 5.7880388  6.45052298
  4.81534641 7.35831353 4.3520441  4.55584145 5.62495704 3.74858054
  4.59647664 3.12810046]
 [3.77889582 3.94454343 2.59447184 4.46393684 3.7571941  4.21418753
  4.11375278 4.6851713  3.98998507 3.23676885 4.53973181 5.35913109
  4.24164123 5.74002393 3.81588358 3.88323363 4.63402675 3.43013819
  3.86199725 2.71263574]
 [3.99136884 4.14003849 3.55784314 4.49000074 4.38329439 5.65769365
  4.22815727 5.39544521 4.96355173 3.41927731 5.93256262 5.12369341
  3.92912918 6.72627348 4.08955557 4.73236522 5.44254809 4.27992319
  4.5173719  3.32348818]
 [4.26805359 5.00617879 3.74007203 5.43399817 5.03502747 5.62603039
  4.8600567  5.46353493 4.78607557 4.07719626 6.15123837 5.9844239
  5.19437574 7.6862292  4.36050822 5.60494358 5.88615565 3.85701771
  5.36024558 3.17493661]
 [4.04100802 3.75543853 2.68002284 4.26844467 4.152182   5.34050129
  4.36370774 5.52061359 5.03454571 3.59604589 5.01579788 5.03933554
  4.26841697 6.40603224 4.0537218  4.35562927 4.98772164 3.87234397
  4.10189591 2.79649112]
 [4.23893136 5.4290815  3.75077832 5.31526326 4.43580616 6.17412115
  5.41353445 5.41744976 5.11878514 5.02387623 5.84854389 6.56775202
  5.23173449 7.86083487 4.97522984 4.67739176 6.13334664 4.91901843
  5.16553436 4.16228894]]
[[4.63702448 4.00488453 3.95522105 4.3789378  4.06176472 6.50679293
  4.77610576 5.18026506 5.43375745 3.68733768 5.44798516 5.53971301
  5.22864341 7.36354183 4.69340147 4.84478703 5.47757704 4.12367848
  5.60415434 3.00976476]
 [4.29925627 5.04363664 2.87654607 4.54102703 4.15759764 4.74121209
  4.63572594 5.12392859 4.47593651 4.33363443 5.06181023 6.06476145
  4.77180514 6.45983505 4.15387961 4.69511556 4.70694237 4.22200305
  4.38602656 3.15149349]
 [4.96381087 5.66338601 4.65574156 5.47686973 5.25968199 7.00427169
  5.86144816 6.12315044 6.32488818 5.09630194 7.20214127 7.44667657
  5.11769343 8.84473486 5.94871368 5.86300717 6.13296376 5.79212848
  6.09076094 4.70590872]
 [3.78902405 3.93362481 4.01241918 3.29401633 3.84989007 4.89406138
  3.78610562 4.35577752 4.30408909 3.13783986 5.08438961 5.0340093
  4.09319394 6.70995704 4.02036172 4.27173153 4.63853685 3.70733642
  4.59210347 3.22371253]
 [3.14900379 3.16740219 2.06164627 3.13969426 3.02967164 3.75436913
  3.3702899  3.75404068 3.47030645 3.1183955  3.86426625 4.02971287
  3.55351106 4.71747241 3.65011727 3.50828188 3.77980197 3.00203675
  3.4424578  2.80047338]
 [3.49162826 4.77443846 3.14653553 4.73107315 4.1584574  5.07381921
  4.50977718 5.11257024 4.45667501 3.5379521  5.46228471 5.12060025
  4.60466859 6.69050065 4.01544832 4.85247315 5.30289004 4.21803987
  4.27673867 3.43547668]
 [3.30197086 3.16063463 3.13435868 3.43208713 2.78414018 4.59130127
  3.72678482 3.83201097 4.14642673 2.56235183 3.89392687 3.96436895
  3.68545876 5.22423285 3.69562161 3.24283721 4.06810341 3.39418248
  4.47197951 2.82054583]
 [4.13832891 5.25431609 3.62557884 4.73876228 4.4127073  5.99077609
  5.11242383 5.61753873 5.70581366 4.2126438  5.09883775 6.01067667
  4.80083055 7.65903446 4.57144045 4.54128483 5.94087051 4.54891694
  5.1315085  3.11247694]
 [5.13521299 5.5608853  5.0593564  5.82321474 5.39704587 6.78302327
  6.48769961 6.70449365 6.59059167 4.03895663 6.85172016 7.30673603
  5.97315875 8.86129836 5.83587961 6.39003051 6.67574901 5.40803169
  6.3885298  4.89191892]
 [3.68493497 3.47206051 3.2780746  4.33465906 4.09016041 4.58848072
  3.36772616 4.6017977  3.63108525 3.05957676 4.85599618 5.09139752
  4.14511414 6.20435562 4.25651063 5.14177892 5.19375251 3.41841606
  4.20871593 3.37121056]
 [4.93498254 6.34455419 4.42132026 5.99520938 5.12706902 6.85305969
  6.43472797 6.73489273 6.32455272 5.66886088 5.96587062 7.19850398
  6.20743213 9.21207209 5.2608264  5.8772283  6.43761049 5.31557252
  6.51244106 4.6404681 ]
 [3.33581865 4.18695016 2.87540272 3.80991169 3.93366442 4.76989052
  4.61857207 4.5279083  4.63445457 3.04085352 5.24580175 4.84231847
  4.46864152 6.10288021 3.49837375 4.10939816 5.11703362 3.78477507
  3.66764909 2.93006523]
 [3.30872372 3.92886087 3.45767273 4.15103667 3.81246504 4.13924995
  4.02071017 4.43012935 3.9240151  2.85147715 4.87761493 5.58511617
  3.79142515 5.847782   3.71523084 4.22465862 3.96365732 3.44447769
  4.37304689 2.98120635]
 [3.84089899 4.06287974 2.70315815 4.37998991 4.18930731 5.3761865
  5.07630943 5.38302457 4.96539644 4.44692224 5.25237338 5.11202151
  4.82402705 6.47816027 4.94809273 4.91040691 5.02022561 3.6581127
  4.36661232 3.77809113]
 [3.9659709  4.40100824 3.09104657 5.13349265 4.89710378 5.21532118
  4.69109561 5.50864695 4.86928118 4.32130978 5.7880388  6.45052298
  4.81534641 7.35831353 4.3520441  4.55584145 5.62495704 3.74858054
  4.59647664 3.12810046]
 [3.77889582 3.94454343 2.59447184 4.46393684 3.7571941  4.21418753
  4.11375278 4.6851713  3.98998507 3.23676885 4.53973181 5.35913109
  4.24164123 5.74002393 3.81588358 3.88323363 4.63402675 3.43013819
  3.86199725 2.71263574]
 [3.99136884 4.14003849 3.55784314 4.49000074 4.38329439 5.65769365
  4.22815727 5.39544521 4.96355173 3.41927731 5.93256262 5.12369341
  3.92912918 6.72627348 4.08955557 4.73236522 5.44254809 4.27992319
  4.5173719  3.32348818]
 [4.26805359 5.00617879 3.74007203 5.43399817 5.03502747 5.62603039
  4.8600567  5.46353493 4.78607557 4.07719626 6.15123837 5.9844239
  5.19437574 7.6862292  4.36050822 5.60494358 5.88615565 3.85701771
  5.36024558 3.17493661]
 [4.04100802 3.75543853 2.68002284 4.26844467 4.152182   5.34050129
  4.36370774 5.52061359 5.03454571 3.59604589 5.01579788 5.03933554
  4.26841697 6.40603224 4.0537218  4.35562927 4.98772164 3.87234397
  4.10189591 2.79649112]
 [4.23893136 5.4290815  3.75077832 5.31526326 4.43580616 6.17412115
  5.41353445 5.41744976 5.11878514 5.02387623 5.84854389 6.56775202
  5.23173449 7.86083487 4.97522984 4.67739176 6.13334664 4.91901843
  5.16553436 4.16228894]]
```

```
# time testing
#lets compute the numpy matrix products as basis for comparison (even
though we cant expect to beat this)

def numpy_matrix_product(mat1,mat2):
    assert mat1.shape[1]== mat2.shape[0]
    return mat1@mat2




# Time the custom matrix multiplication
execution_time1 = timeit.timeit('slow_matrix_product(matrix1,
matrix2)', globals=globals(), number=1000)

# Time the NumPy matrix multiplication using @ operator
execution_time2 = timeit.timeit('numpy_matrix_product(matrix1,
matrix2)', globals=globals(), number=1000)

print(f"Execution time for slow_matrix_product: {execution_time1}
seconds")
print(f"Execution time for NumPy matrix multiplication (@):
{execution_time2} seconds")
print(f"performance difference factor:
{execution_time1/execution_time2}")
```

```
Execution time for slow_matrix_product: 3.765986373 seconds
Execution time for NumPy matrix multiplication (@):
0.003223177000000632 seconds
performance difference factor: 1168.4081801896891
```

Part 1: a better function

Write your own function called faster_matrix_product that computes the product of two matrices more efficiently than slow_matrix_product. Your function may use functions from Numpy (eg np.dot or @) to complete part of its calculation, but your function should not use np.dot or @ to compute the full matrix-matrix product.

Note: In checking my work, I realise this function should have been called 'faster_matrix_product' but it takes a very long time to run the following computations for my laptop, therefore I hope you will forgive me for not changing the name, so it is fast_matrix_product(). Thank you

```
def fast_matrix_product(mat1, mat2):
    """Multiply two matrices.The fast way!"""
    assert mat1.shape[1] == mat2.shape[0]

    output = np.zeros((mat1.shape[0],mat2.shape[1]))
    #double nested loop over columns and rows
    for i in range(mat1.shape[1]):
        for j in range(mat2.shape[0]):
            output[i,j]=np.dot(mat1[i,:],mat2[:,j])
    return output


out=fast_matrix_product(matrix1,matrix2)
print(out)
```

```
[[4.63702448 4.00488453 3.95522105 4.3789378  4.06176472 6.50679293
  4.77610576 5.18026506 5.43375745 3.68733768 5.44798516 5.53971301
  5.22864341 7.36354183 4.69340147 4.84478703 5.47757704 4.12367848
  5.60415434 3.00976476]
 [4.29925627 5.04363664 2.87654607 4.54102703 4.15759764 4.74121209
  4.63572594 5.12392859 4.47593651 4.33363443 5.06181023 6.06476145
  4.77180514 6.45983505 4.15387961 4.69511556 4.70694237 4.22200305
  4.38602656 3.15149349]
 [4.96381087 5.66338601 4.65574156 5.47686973 5.25968199 7.00427169
  5.86144816 6.12315044 6.32488818 5.09630194 7.20214127 7.44667657
  5.11769343 8.84473486 5.94871368 5.86300717 6.13296376 5.79212848
  6.09076094 4.70590872]
 [3.78902405 3.93362481 4.01241918 3.29401633 3.84989007 4.89406138
  3.78610562 4.35577752 4.30408909 3.13783986 5.08438961 5.0340093
  4.09319394 6.70995704 4.02036172 4.27173153 4.63853685 3.70733642
  4.59210347 3.22371253]
 [3.14900379 3.16740219 2.06164627 3.13969426 3.02967164 3.75436913
  3.3702899  3.75404068 3.47030645 3.1183955  3.86426625 4.02971287
  3.55351106 4.71747241 3.65011727 3.50828188 3.77980197 3.00203675
  3.4424578  2.80047338]
 [3.49162826 4.77443846 3.14653553 4.73107315 4.1584574  5.07381921
  4.50977718 5.11257024 4.45667501 3.5379521  5.46228471 5.12060025
  4.60466859 6.69050065 4.01544832 4.85247315 5.30289004 4.21803987
  4.27673867 3.43547668]
 [3.30197086 3.16063463 3.13435868 3.43208713 2.78414018 4.59130127
  3.72678482 3.83201097 4.14642673 2.56235183 3.89392687 3.96436895
  3.68545876 5.22423285 3.69562161 3.24283721 4.06810341 3.39418248
  4.47197951 2.82054583]
 [4.13832891 5.25431609 3.62557884 4.73876228 4.4127073  5.99077609
  5.11242383 5.61753873 5.70581366 4.2126438  5.09883775 6.01067667
  4.80083055 7.65903446 4.57144045 4.54128483 5.94087051 4.54891694
  5.1315085  3.11247694]
 [5.13521299 5.5608853  5.0593564  5.82321474 5.39704587 6.78302327
  6.48769961 6.70449365 6.59059167 4.03895663 6.85172016 7.30673603
  5.97315875 8.86129836 5.83587961 6.39003051 6.67574901 5.40803169
  6.3885298  4.89191892]
 [3.68493497 3.47206051 3.2780746  4.33465906 4.09016041 4.58848072
  3.36772616 4.6017977  3.63108525 3.05957676 4.85599618 5.09139752
  4.14511414 6.20435562 4.25651063 5.14177892 5.19375251 3.41841606
  4.20871593 3.37121056]
 [4.93498254 6.34455419 4.42132026 5.99520938 5.12706902 6.85305969
  6.43472797 6.73489273 6.32455272 5.66886088 5.96587062 7.19850398
  6.20743213 9.21207209 5.2608264  5.8772283  6.43761049 5.31557252
  6.51244106 4.6404681 ]
 [3.33581865 4.18695016 2.87540272 3.80991169 3.93366442 4.76989052
  4.61857207 4.5279083  4.63445457 3.04085352 5.24580175 4.84231847
  4.46864152 6.10288021 3.49837375 4.10939816 5.11703362 3.78477507
  3.66764909 2.93006523]
 [3.30872372 3.92886087 3.45767273 4.15103667 3.81246504 4.13924995
  4.02071017 4.43012935 3.9240151  2.85147715 4.87761493 5.58511617
  3.79142515 5.847782   3.71523084 4.22465862 3.96365732 3.44447769
  4.37304689 2.98120635]
 [3.84089899 4.06287974 2.70315815 4.37998991 4.18930731 5.3761865
  5.07630943 5.38302457 4.96539644 4.44692224 5.25237338 5.11202151
  4.82402705 6.47816027 4.94809273 4.91040691 5.02022561 3.6581127
  4.36661232 3.77809113]
 [3.9659709  4.40100824 3.09104657 5.13349265 4.89710378 5.21532118
  4.69109561 5.50864695 4.86928118 4.32130978 5.7880388  6.45052298
  4.81534641 7.35831353 4.3520441  4.55584145 5.62495704 3.74858054
  4.59647664 3.12810046]
 [3.77889582 3.94454343 2.59447184 4.46393684 3.7571941  4.21418753
  4.11375278 4.6851713  3.98998507 3.23676885 4.53973181 5.35913109
  4.24164123 5.74002393 3.81588358 3.88323363 4.63402675 3.43013819
  3.86199725 2.71263574]
 [3.99136884 4.14003849 3.55784314 4.49000074 4.38329439 5.65769365
  4.22815727 5.39544521 4.96355173 3.41927731 5.93256262 5.12369341
  3.92912918 6.72627348 4.08955557 4.73236522 5.44254809 4.27992319
  4.5173719   3.32348818]
 [4.26805359 5.00617879 3.74007203 5.43399817 5.03502747 5.62603039
  4.8600567  5.46353493 4.78607557 4.07719626 6.15123837 5.9844239
  5.19437574 7.6862292  4.36050822 5.60494358 5.88615565 3.85701771
  5.36024558 3.17493661]
 [4.04100802 3.75543853 2.68002284 4.26844467 4.152182   5.34050129
  4.36370774 5.52061359 5.03454571 3.59604589 5.01579788 5.03933554
  4.26841697 6.40603224 4.0537218  4.35562927 4.98772164 3.87234397
  4.10189591 2.79649112]
 [4.23893136 5.4290815  3.75077832 5.31526326 4.43580616 6.17412115
  5.41353445 5.41744976 5.11878514 5.02387623 5.84854389 6.56775202
  5.23173449 7.86083487 4.97522984 4.67739176 6.13334664 4.91901843
  5.16553436 4.16228894]]

# lets time the custom matrix multiplication
execution_time1 = timeit.timeit('slow_matrix_product(matrix1,
matrix2)', globals=globals(), number=1000)

# lets time the NumPy matrix multiplication using @ operator
execution_time2 = timeit.timeit('fast_matrix_product(matrix1,
matrix2)', globals=globals(), number=1000)

print(f"Execution time for slow_matrix_product: {execution_time1}
seconds") # printing out all the times
print(f"Execution time for fast matrix multiplication (@):
{execution_time2} seconds")
print(f"performance difference factor:
{execution_time1/execution_time2}")

Execution time for slow_matrix_product: 3.736433462999999 seconds
Execution time for fast matrix multiplication (@): 0.7231960219999998
seconds
performance difference factor: 5.166556990547163
```

before you look at the performance of your function, you should check that it is computing the correct results. Write a Python script using an assert statement that checks that your function gives the same result as using @ for random 2 by 2, 3 by 3, 4 by 4, and 5 by 5 matrices.

```
for i in range(2,6): # loop
    #generate random matrices and test using numpy_matrix_product() as
basis for comparison
    matrix1 = np.random.rand(i, i)
    matrix2 = np.random.rand(i, i)

    val_a = numpy_matrix_product(matrix1,matrix2)
    val_b = fast_matrix_product(matrix1,matrix2)
    assert np.allclose(val_a,val_b) # checking they match
    print(f"Test passed for {i}x{i} matrices")
```

```
Test passed for 2x2 matrices
Test passed for 3x3 matrices
Test passed for 4x4 matrices
Test passed for 5x5 matrices
```

In a text box, give two brief reasons (1-2 sentences for each) why your function is better than slow_matrix_product. At least one of your reasons should be related to the time you expect the two functions to take.

Reason 1: The optimised python funciton fast_matrix_product() reduces the number of nested loops from 3 to 2, this has the result of reducing the time complexity of the function from O(n^3) something which is less than O(n^3) but greater than O(n^2), n is the number of rows/columns on the matrix.

Reason 2: The memoray allocation is done in a much more efficient manner in our optimised function. In the slower function the data is appended at two instances within the loop when data is appended to 'column' and 'result', resulting in n^2 appendages (if this is word?) to the data. While in our optimised function the memory is already predefined and pre allocated when we do:output = np.zeros((mat1.shape[0],mat2.shape[1])), so it only happens once.

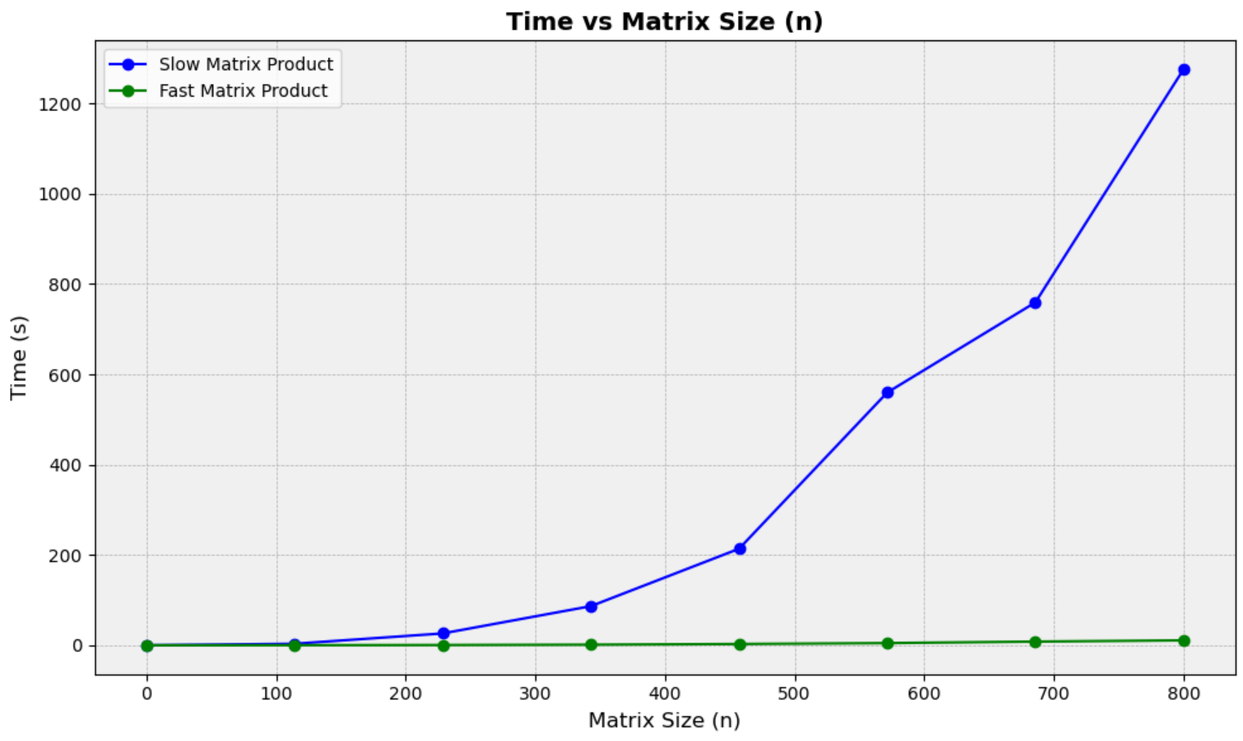
Task 1 part 2

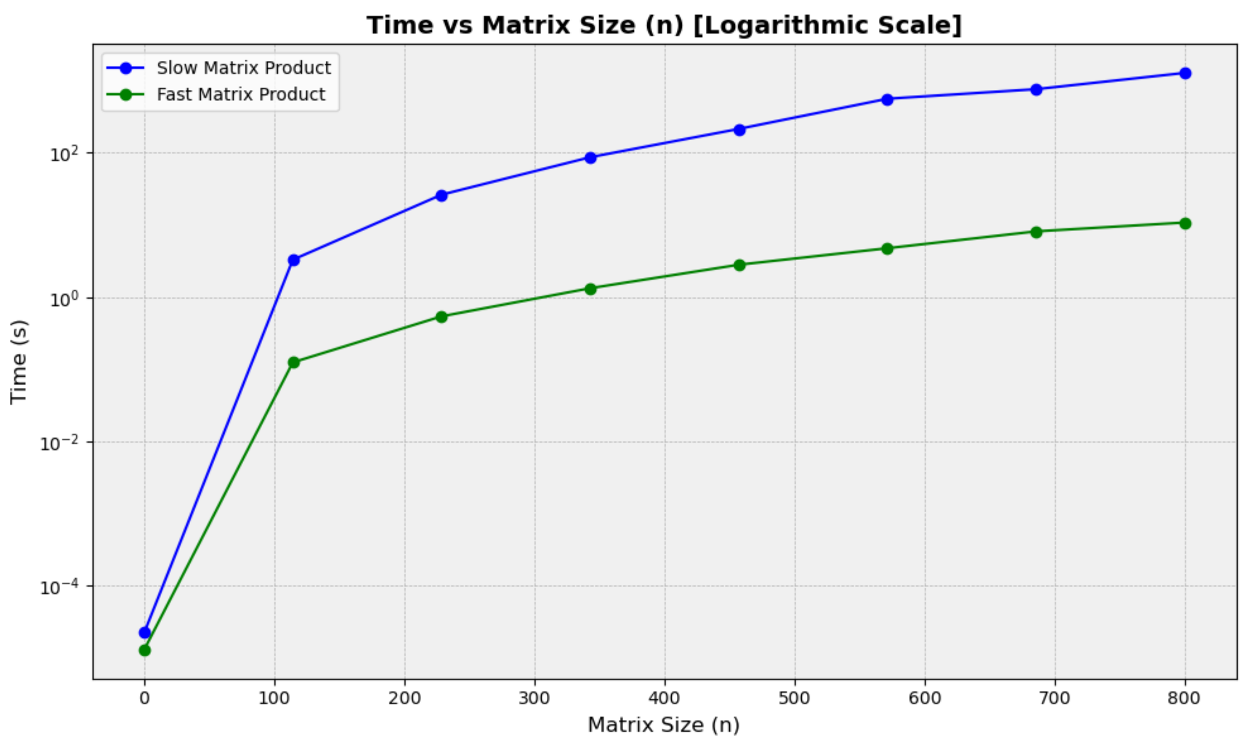
Next, we want to compare the speed of slow_matrix_product and faster_matrix_product. Write a Python script that runs the two functions for matrices of a range of sizes, and use matplotlib to create a plot showing the time taken for different-sized matrices for both functions. You should be able to run the functions for matrices of size up to around 1000 by 1000 (but if you're using an older/slower computer, you may decide to decrease the maximums slightly). You do not need to run your functions for every size between your minimum and maximum but should choose a set of 10-15 values that will give you an informative plot.

```python
#we want to create a range of n sizes to comapre the 2 functions
n_range = np.linspace(0,800,8)

#initialise empty arrays to store values

fast_times = []
slow_times = []



def run_test_into_arr(function, landing_arr, range_):
    '''
    A function while fills a target array with the run times of a 
specified function
    '''
    for value in range(len(range_)):
        #generate testing values for function input
        testmat1, testmat2 =
np.random.rand(int(range_[value]),int(range_[value])),np.random.rand(i
nt(range_[value]),int(range_[value]))
        time = timeit.timeit(lambda: function(testmat1,testmat2),
number=5)
        landing_arr.append(time)







#adding the values to out arrays using the function
run_test_into_arr(slow_matrix_product, slow_times, n_range)
run_test_into_arr(fast_matrix_product, fast_times, n_range)


print(fast_times)
print(slow_times)

#plotting
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(n_range, slow_times, 'o-', label="Slow Matrix Product",
color='blue')  # colour and styling
ax.plot(n_range, fast_times, 'o-', label="Fast Matrix Product",
color='green')
ax.set_title("Time vs Matrix Size (n)", fontsize=14,
fontweight='bold')
ax.set_xlabel("Matrix Size (n)", fontsize=12)
ax.set_ylabel("Time (s)", fontsize=12)
ax.legend(loc='best', fontsize=10)
ax.grid(True, linestyle='--', linewidth=0.5)  #  grid style
ax.set_facecolor('#f0f0f0')  #  background color
plt.tight_layout()
plt.show()

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(n_range, slow_times, 'o-', label="Slow Matrix Product",
color='blue')  #  color and style
ax.plot(n_range, fast_times, 'o-', label="Fast Matrix Product",
color='green')  #  color and style
ax.set_title("Time vs Matrix Size (n) [Logarithmic Scale]",
fontsize=14, fontweight='bold')  #  title font size / weight
ax.set_xlabel("Matrix Size (n)", fontsize=12)  #  label font size
ax.set_ylabel("Time (s)", fontsize=12)  #  label font size
ax.set_yscale("log")  # log scale for y-axis
ax.legend(loc='best', fontsize=10)
ax.grid(True, which="both", linestyle='--', linewidth=0.5)  #  grid
style
```

```python
ax.set_facecolor('#f0f0f0')  #  background color
plt.tight_layout()
plt.show()
```

```
[1.308899982177536e-05, 0.12466573400024572, 0.5413208710001527,
1.3279668300001504, 2.8089193399996475, 4.756242208000003,
8.141011286999856, 10.797326298999906]
[2.2458000000113998e-05, 3.2958096670000003, 26.216516789000003,
86.612758266, 213.701780115, 559.586032729, 759.175531481,
1275.260583682]
```





Please note: I tried to run n=1000, however this was taking too long for my old laptop, after trying to run this all night i settled on n=800

Part 2: speeding it up with Numba

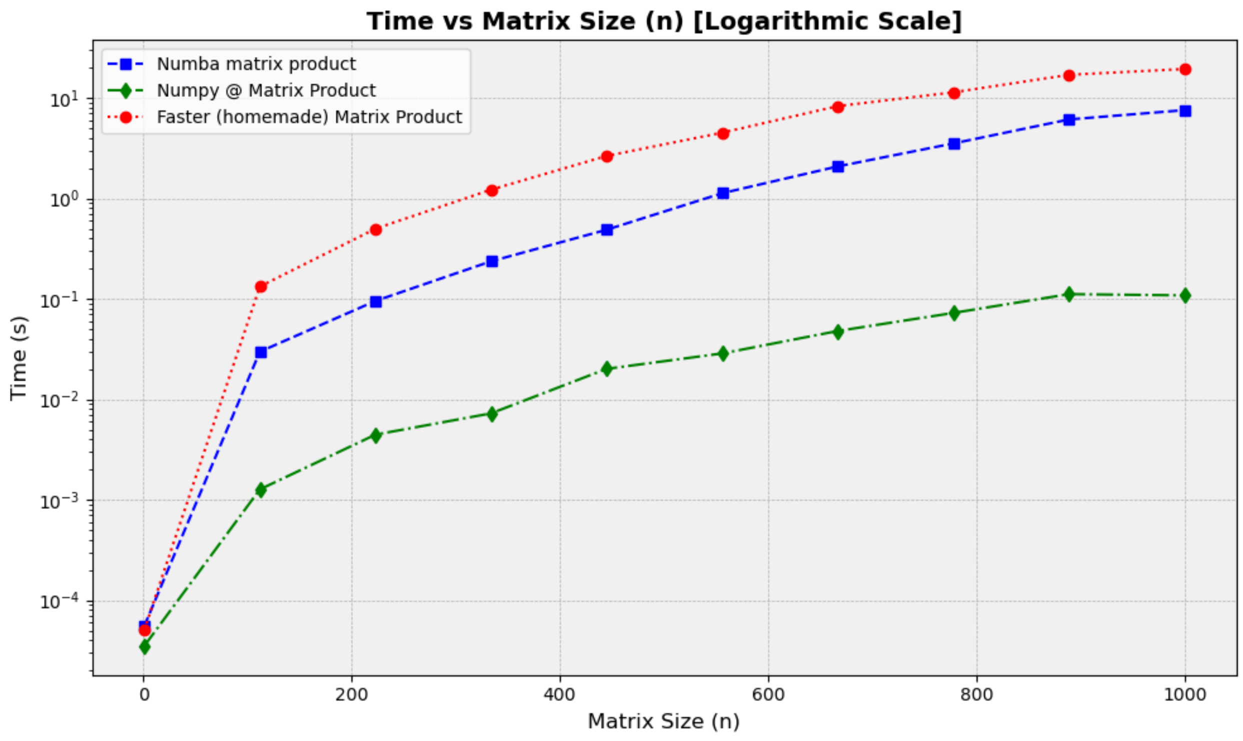
In the second part of this assignment, you're going to use Numba to speed up your function.

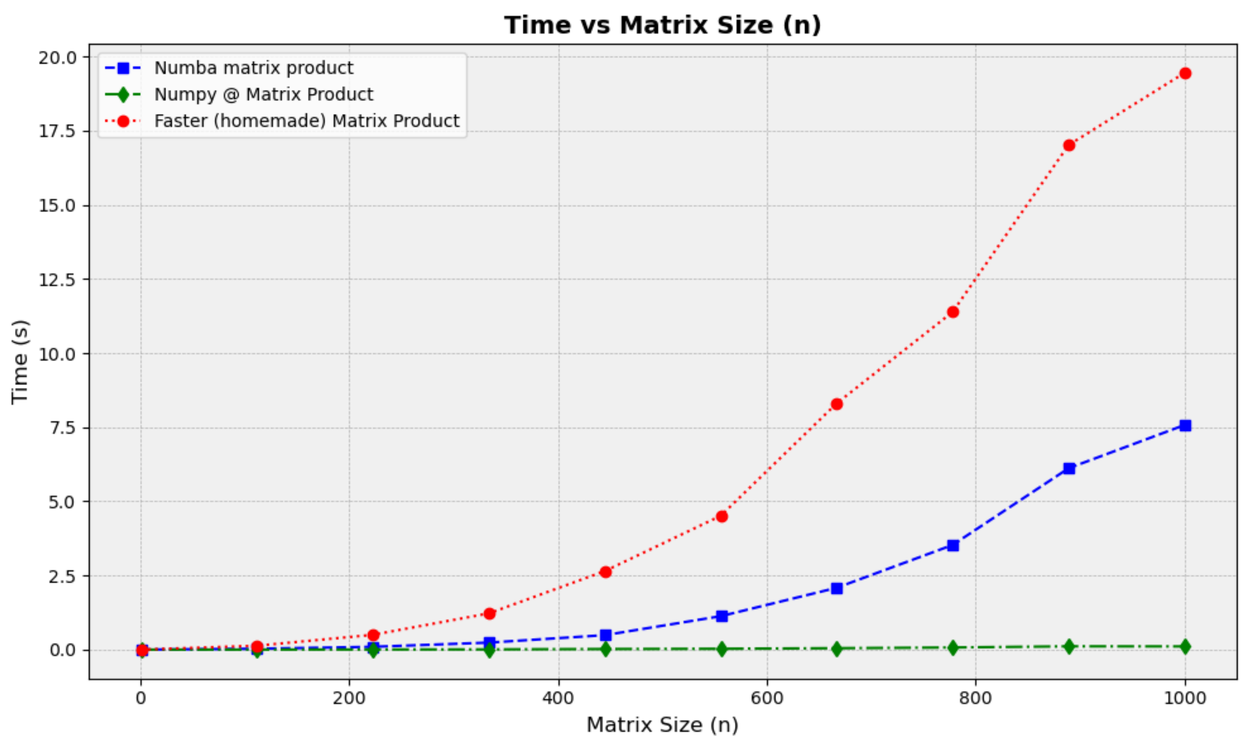
Create a copy of your function faster_matrix_product that is just-in-time (JIT) compiled using Numba.

```python
from numba import njit

@njit
def numba_matrix_product(mat1, mat2):
    """Multiply two matrices.The fast way! This time using numba"""
    assert mat1.shape[1] == mat2.shape[0]

    output = np.zeros((mat1.shape[0],mat2.shape[1]))
    #double nested loop over columns and rows
    for i in range(mat1.shape[1]):
        for j in range(mat2.shape[0]):
            output[i,j]=np.dot(mat1[i,:],mat2[:,j])
    return output
```

To demonstrate the speed improvement acheived by using Numba, make a plot (similar to that you made in the first part) that shows the times taken to multiply matrices using faster_matrix_product, faster_matrix_product with Numba JIT compilation and Numpy (@). Numpy's matrix-matrix multiplication is highly optimised, so you should not expect to be as fast is it.

```
numba_time_arr2=[]
numpy_time_arr2=[]
faster_time_arr2=[]

#using the function that we have already createated to load values
into these
n_range2=np.linspace(1,1000,10)
run_test_into_arr(numba_matrix_product,numba_time_arr2,n_range2)
run_test_into_arr(numpy_matrix_product,numpy_time_arr2,n_range2)
run_test_into_arr(fast_matrix_product, faster_time_arr2, n_range2)


fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(n_range2, numba_time_arr2, 's--', label="Numba matrix
product", color='blue')
ax.plot(n_range2, numpy_time_arr2, 'd-.', label="Numpy @ Matrix
Product", color='green')
ax.plot(n_range2, faster_time_arr2, 'o:', label="Faster (homemade)
Matrix Product", color='red')
ax.set_title("Time vs Matrix Size (n) [Logarithmic Scale]",
fontsize=14, fontweight='bold')
ax.set_xlabel("Matrix Size (n)", fontsize=12)
ax.set_ylabel("Time (s)", fontsize=12)
ax.set_yscale('log')
ax.legend(loc='best', fontsize=10)
ax.grid(True, linestyle='--', linewidth=0.5)
ax.set_facecolor('#f0f0f0')

plt.tight_layout()
plt.show()

fig, ax = plt.subplots(figsize=(10, 6))

ax.plot(n_range2, numba_time_arr2, 's--', label="Numba matrix
product", color='blue')
ax.plot(n_range2, numpy_time_arr2, 'd-.', label="Numpy @ Matrix
Product", color='green')
ax.plot(n_range2, faster_time_arr2, 'o:', label="Faster (homemade)
Matrix Product", color='red')

ax.set_title("Time vs Matrix Size (n)", fontsize=14,
fontweight='bold')
ax.set_xlabel("Matrix Size (n)", fontsize=12)
ax.set_ylabel("Time (s)", fontsize=12)
# Removing logarithmic scale
# ax.set_yscale('log')

ax.legend(loc='best', fontsize=10)
ax.grid(True, linestyle='--', linewidth=0.5)
ax.set_facecolor('#f0f0f0')

plt.tight_layout()
plt.show()
```



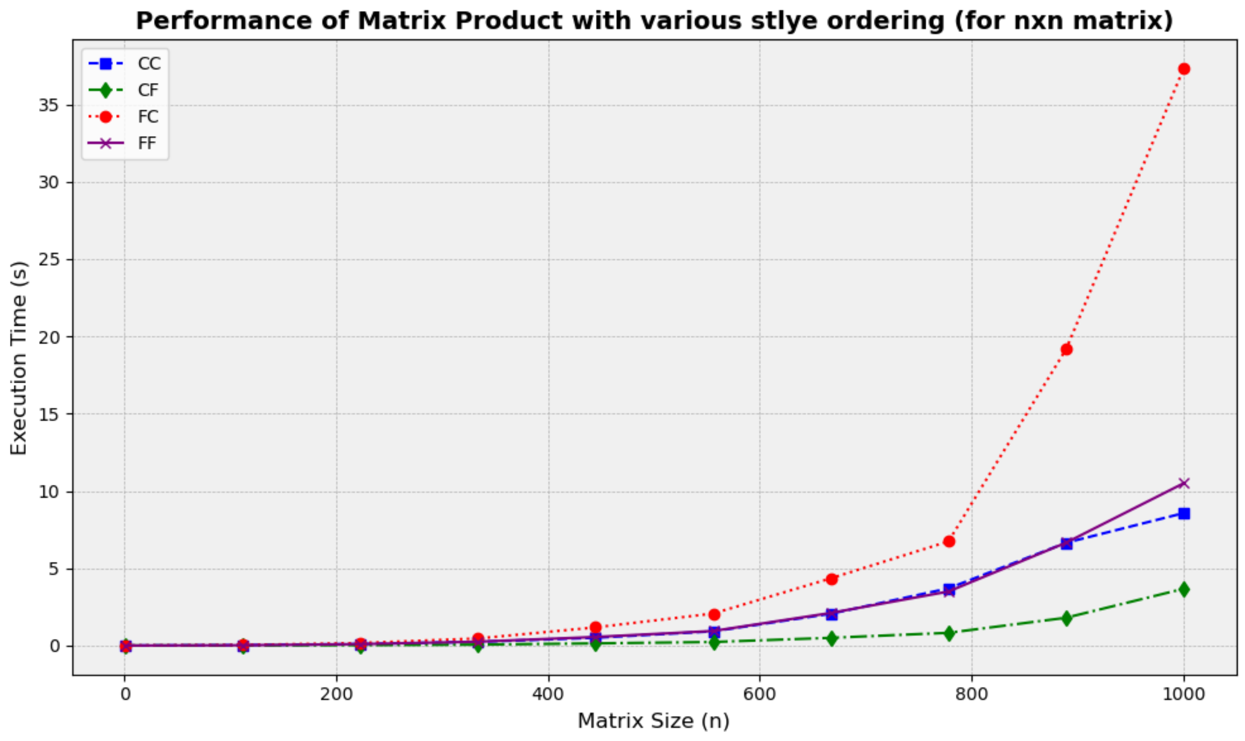


```
#thats more like it, we probably shouldnt use 0x0 matricies anyway
```

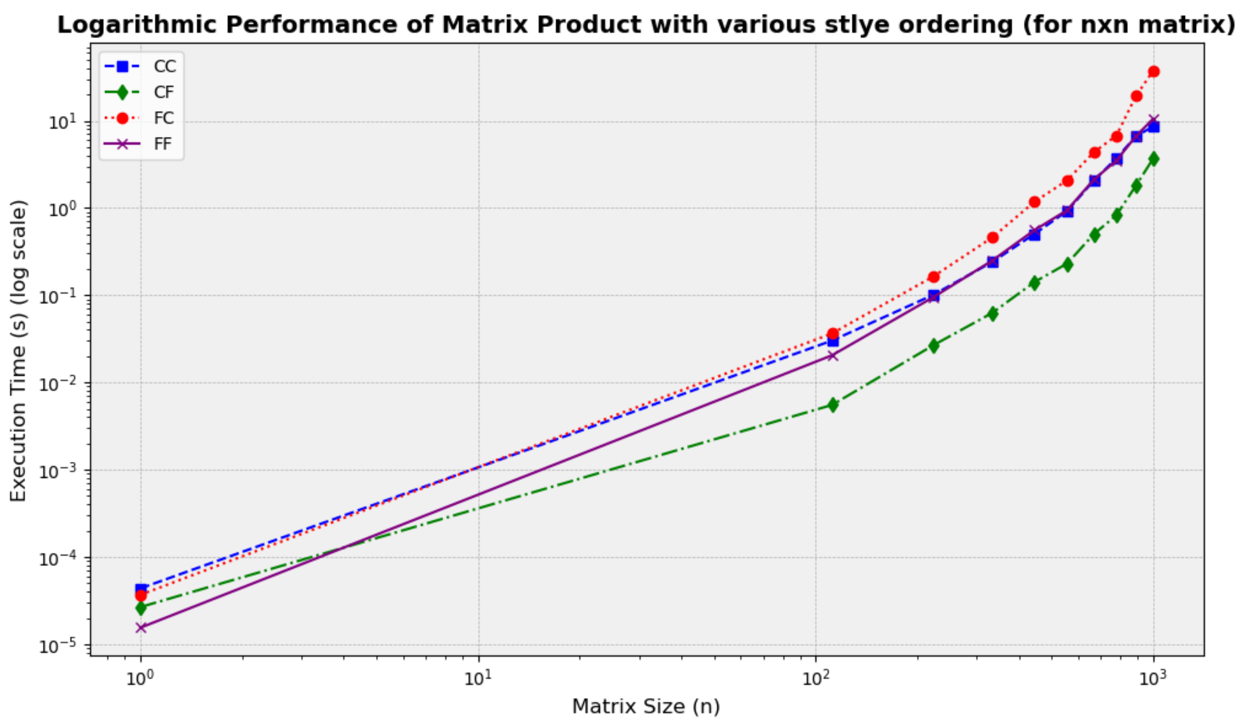
Task 2 part b

Make a plot that compares the times taken by your JIT compiled function when the inputs have different combinations of C-style and Fortran-style ordering (ie the plot should have lines for when both inputs are C-style, when the first is C-style and the second is Fortran-style, and so on).

```
#setting up matrices with different style orderings
# CC, then C fortran, then Fortran C, then fortran fortran?

c_c=[]
c_f=[]
f_c=[]
f_f=[]




dimensions=[]
for i in n_range2: #using the same range as in Task 1
    c1,c2 =
np.random.rand(int(i),int(i)),np.random.rand(int(i),int(i))
    f1,f2 = np.asfortranarray(c1),np.asfortranarray(c2)

    _cc =
timeit.timeit('numba_matrix_product(c1,c2)',globals=globals(),number=5
) #using the numba jiit function
    _cf =
timeit.timeit('numba_matrix_product(c1,f2)',globals=globals(),number=5
)
    _fc =
timeit.timeit('numba_matrix_product(f1,c2)',globals=globals(),number=5
)
    _ff =
timeit.timeit('numba_matrix_product(f1,f2)',globals=globals(),number=5
)

    dimensions.append(i) #appending steps
    c_c.append(_cc) #appending time values for step i
    c_f.append(_cf)
    f_c.append(_fc)
    f_f.append(_ff)



fig, ax = plt.subplots(figsize=(10, 6)) #plotting

ax.plot(dimensions, c_c, 's--', label="CC", color='blue')
ax.plot(dimensions, c_f, 'd-.', label="CF", color='green')
ax.plot(dimensions, f_c, 'o:', label="FC", color='red')
ax.plot(dimensions, f_f, 'x-', label="FF", color='purple')

ax.set_title("Performance of Matrix Product with various stlye
ordering (for nxn matrix)", fontsize=14, fontweight='bold')
ax.set_xlabel("Matrix Size (n)", fontsize=12)
ax.set_ylabel("Execution Time (s)", fontsize=12)

ax.legend(loc='best', fontsize=10)
ax.grid(True, linestyle='--', linewidth=0.5)
ax.set_facecolor('#f0f0f0')

plt.tight_layout()
plt.show()
```



```python
fig, ax = plt.subplots(figsize=(10, 6)) #plotting log version of graph
above
ax.plot(dimensions, c_c, 's--', label="CC", color='blue')
ax.plot(dimensions, c_f, 'd-.', label="CF", color='green')
ax.plot(dimensions, f_c, 'o:', label="FC", color='red')
ax.plot(dimensions, f_f, 'x-', label="FF", color='purple')
ax.set_title("Logarithmic Performance of Matrix Product with various
stlye ordering (for nxn matrix)", fontsize=14, fontweight='bold')
ax.set_xlabel("Matrix Size (n)", fontsize=12)
ax.set_ylabel("Execution Time (s) (log scale)", fontsize=12)

ax.set_xscale('log')
ax.set_yscale('log')

ax.legend(loc='best', fontsize=10)
ax.grid(True, linestyle='--', linewidth=0.5)
ax.set_facecolor('#f0f0f0')

plt.tight_layout()
plt.show()
```



Focusing on the fact that it is more efficient to access memory that is close to previous accesses, comment (in 1-2 sentences) on why one of these orderings appears to be fastest that the others. (Numba can do a lot of different things when compiling code, so depending on your function there may or may not be a large difference: if there is little change in speeds for your function, you can comment on which ordering you might expect to be faster and why, but conclude that Numba is doing something more advanced.)

The reason behind this difference is due to the memory layout and how the data is stored. If we look at the C-style ordering, the data is stored row-wise, while in Fortran-ordering, the data is stored column-wise. Since matrix multiplication involves taking the dot product of rows and columns of the first and second matrix respectively, the memory access in this case is more optimal. The nature of both memory accesses will theremfore minimise distance between the bits accesses in the memory, increasing performance. This explains why CF ordering is fastest, while FC memory is the slowest as in this case the data layout on either side of the multiplication sign is opposite to the optimal case.

One supporting piece of evidence to this idea is the fact that the log plot shows the same gradients, i.e the algorithm's complexity remains the same, the differences in runtime are due to the varying memory layouts. The underlying mathematical operations remain equivalent, but in this, case speed is influenced by how efficiently memory is accessed during computation. As can be seen by the plot above, this manifests itself in a different interept value on the Execution time axis.