Part 1: a better function

```
import numpy as np


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(5, 5)
matrix2 = np.random.rand(5, 5)

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

```
[[0.65572564 1.34823741 1.44917372 0.64950634 1.32139172]
 [0.45483369 1.43954375 1.36846064 0.57593471 1.32083223]
 [0.57671074 2.03573916 1.50794962 1.15231377 1.57457324]
 [0.45112284 1.23617879 1.01955828 0.43830564 0.81670263]
 [0.66827802 1.82817658 1.30367827 0.86714479 1.2632344 ]]
[[0.65572564 1.34823741 1.44917372 0.64950634 1.32139172]
 [0.45483369 1.43954375 1.36846064 0.57593471 1.32083223]
 [0.57671074 2.03573916 1.50794962 1.15231377 1.57457324]
 [0.45112284 1.23617879 1.01955828 0.43830564 0.81670263]
 [0.66827802 1.82817658 1.30367827 0.86714479 1.2632344 ]]
```

```
def split(mat):
    row, col = mat.shape
    r, c = row // 2, col // 2
    return mat[:r, :c], mat[:r, c:], mat[r:, :c], mat[r:, c:]


def faster_matrix_product(mat1, mat2):
    assert mat1.shape[1] == mat2.shape[0]

    flag = 0
    if mat1.shape[0] % 2 != 0:
        temp = np.zeros((1, mat1.shape[0]))
        mat1 = np.vstack((temp, mat1))
        mat2 = np.hstack((temp.T, mat2))
```

```
            temp = np.zeros((mat1.shape[0], 1))
            mat1 = np.hstack((temp, mat1))
            mat2 = np.vstack((temp.T, mat2))

            flag = 1


        a11, a12, a21, a22 = split(mat1)
        b11, b12, b21, b22 = split(mat2)

        s1 = a21 + a22
        s2 = s1 - a11
        s3 = a11 - a21
        s4 = a12 - s2

        t1 = b12 - b11
        t2 = b22 - t1
        t3 = b22 - b12
        t4 = t2 - b21

        m1 = np.dot(a11, b11)
        m2 = np.dot(a12, b21)
        m3 = np.dot(s4, b22)
        m4 = np.dot(a22, t4)
        m5 = np.dot(s1, t1)
        m6 = np.dot(s2, t2)
        m7 = np.dot(s3, t3)

        u1 = m1 + m2
        u2 = m1 + m6
        u3 = u2 + m7
        u4 = u2 + m5
        u5 = u4 + m3
        u6 = u3 - m4
        u7 = u3 + m5

        result = np.hstack((np.vstack((u1, u6)), np.vstack((u5, u7))))

        if flag == 1:
            result = result[1:, 1:]

        return result

A = np.random.rand(2, 2)
B = np.random.rand(2, 2)

assert np.allclose(A @ B, faster_matrix_product(A, B))


C = np.random.rand(3, 3)
D = np.random.rand(3, 3)

assert np.allclose(C @ D, faster_matrix_product(C, D))
```

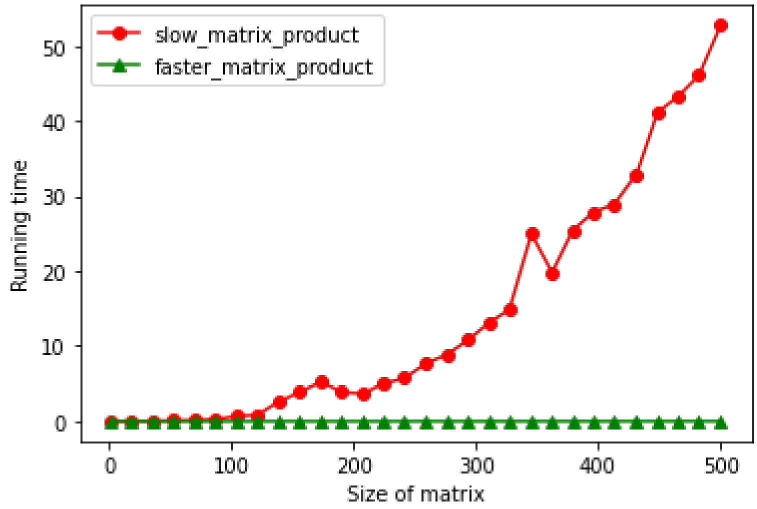
```
E = np.random.rand(4, 4)
F = np.random.rand(4, 4)

assert np.allclose(E @ F, faster_matrix_product(E, F))

H = np.random.rand(5, 5)
I = np.random.rand(5, 5)

assert np.allclose(H @ I, faster_matrix_product(H, I))
```

There are two reasons why the function of faster_matrix_product better than the function of slow_matrix_product:

1. Firstly, there are three loops in slow_matrix_product, and the time complexity is $O(n^3)$. However, in faster_matrix_product, it only used 7 multiplication operations, and 15 addition or subtraction operations, and its time complexity is $O(n^{2.4})$.
2. Secondly, the function of faster_matrix_product uses some functions from Numpy to complete part of its calculation, and those functions make faster_matrix_product faster than slow_matrix_product, what only uses normal operations.

```
from timeit import timeit


def slow_time_counter(mat1, mat2):
    slow_matrix_product(mat1, mat2)


def faster_time_counter(mat1, mat2):
    faster_matrix_product(mat1, mat2)


import matplotlib.pylab as plt
%matplotlib inline


x = np.linspace(1, 500, 30, dtype=np.int32)
y_slow = []
y_faster = []
for i in x:
    mat_1 = np.random.rand(i, i)
    mat_2 = np.random.rand(i, i)
    y_slow.append(timeit(lambda: slow_time_counter(mat_1, mat_2), number=1))
    y_faster.append(timeit(lambda: faster_time_counter(mat_1, mat_2), number=1))


plt.plot(x, y_slow, "ro-")
plt.plot(x, y_faster, "g^-")

plt.xlabel("Size of matrix")
plt.ylabel("Running time")
plt.legend(["slow_matrix_product", "faster_matrix_product"])
```

```
<matplotlib.legend.Legend at 0x7ff47aba7b90>
```

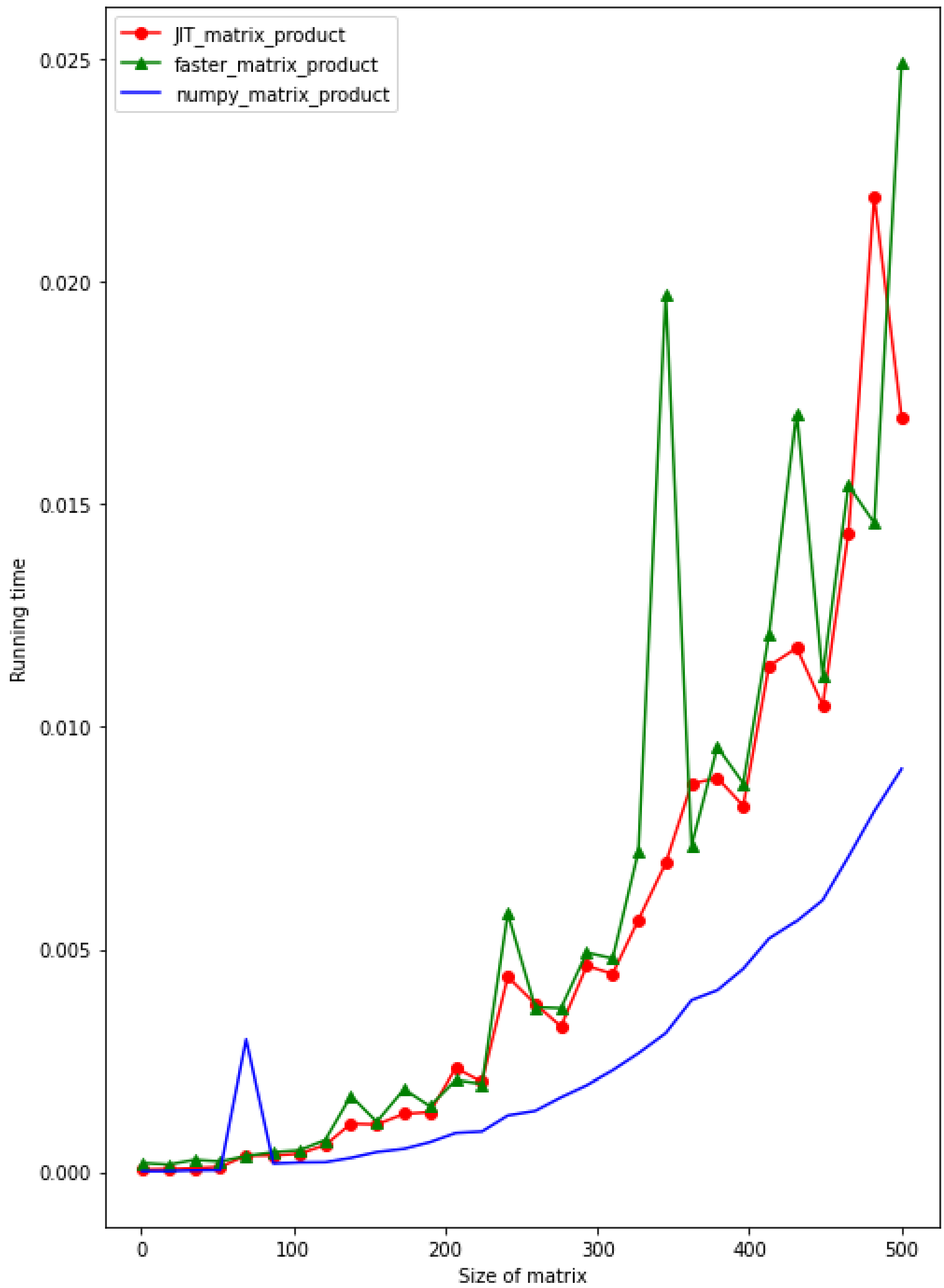


Part 2: speeding it up with Numba

```
import numba


@numba.njit
def split_copy(mat):
    row, col = mat.shape
    r, c = row // 2, col // 2
    return mat[:r, :c], mat[:r, c:], mat[r:, :c], mat[r:, c:]


@numba.njit
def faster_matrix_product_copy(mat1, mat2):
    assert mat1.shape[1] == mat2.shape[0]

    flag = 0
    if mat1.shape[0] % 2 != 0:
        temp = np.zeros((1, mat1.shape[0]))
        mat1 = np.vstack((temp, mat1))
        mat2 = np.hstack((temp.T, mat2))

        temp = np.zeros((mat1.shape[0], 1))
        mat1 = np.hstack((temp, mat1))
        mat2 = np.vstack((temp.T, mat2))

        flag = 1


    a11, a12, a21, a22 = split_copy(mat1)
    b11, b12, b21, b22 = split_copy(mat2)

    s1 = a21 + a22
    s2 = s1 - a11
    s3 = a11 - a21
    s4 = a12 - s2

    t1 = b12 - b11
```

```
        t2 = b22 - t1
        t3 = b22 - b12
        t4 = t2 - b21

        m1 = np.dot(a11, b11)
        m2 = np.dot(a12, b21)
        m3 = np.dot(s4, b22)
        m4 = np.dot(a22, t4)
        m5 = np.dot(s1, t1)
        m6 = np.dot(s2, t2)
        m7 = np.dot(s3, t3)

        u1 = m1 + m2
        u2 = m1 + m6
        u3 = u2 + m7
        u4 = u2 + m5
        u5 = u4 + m3
        u6 = u3 - m4
        u7 = u3 + m5

        result = np.hstack((np.vstack((u1, u6)), np.vstack((u5, u7))))

        if flag == 1:
                result = result[1:, 1:]

        return result


def numpy_time_counter(mat1, mat2):
        mat1 @ mat2


def JIT_time_counter(mat1, mat2):
        faster_matrix_product_copy(mat1, mat2)


x = np.linspace(1, 500, 30, dtype=np.int32)
y_JIT = []
y_faster = []
y_numpy = []
for i in x:
        mat_1 = np.random.rand(i, i)
        mat_2 = np.random.rand(i, i)
        y_JIT.append(timeit(lambda: JIT_time_counter(mat_1, mat_2), number=1))
        y_faster.append(timeit(lambda: faster_time_counter(mat_1, mat_2), number=1))
        y_numpy.append(timeit(lambda: numpy_time_counter(mat_1, mat_2), number=1))


plt.figure(figsize=(8,12))
plt.plot(x, y_JIT, "ro-")
plt.plot(x, y_faster, "g^-")
plt.plot(x, y_numpy, "b-")

plt.xlabel("Size of matrix")
plt.ylabel("Running time")
plt.legend(["JIT_matrix_product", "faster_matrix_product", "numpy_matrix_product"])
```

```
<matplotlib.legend.Legend at 0x7ff467fcfc50>
```

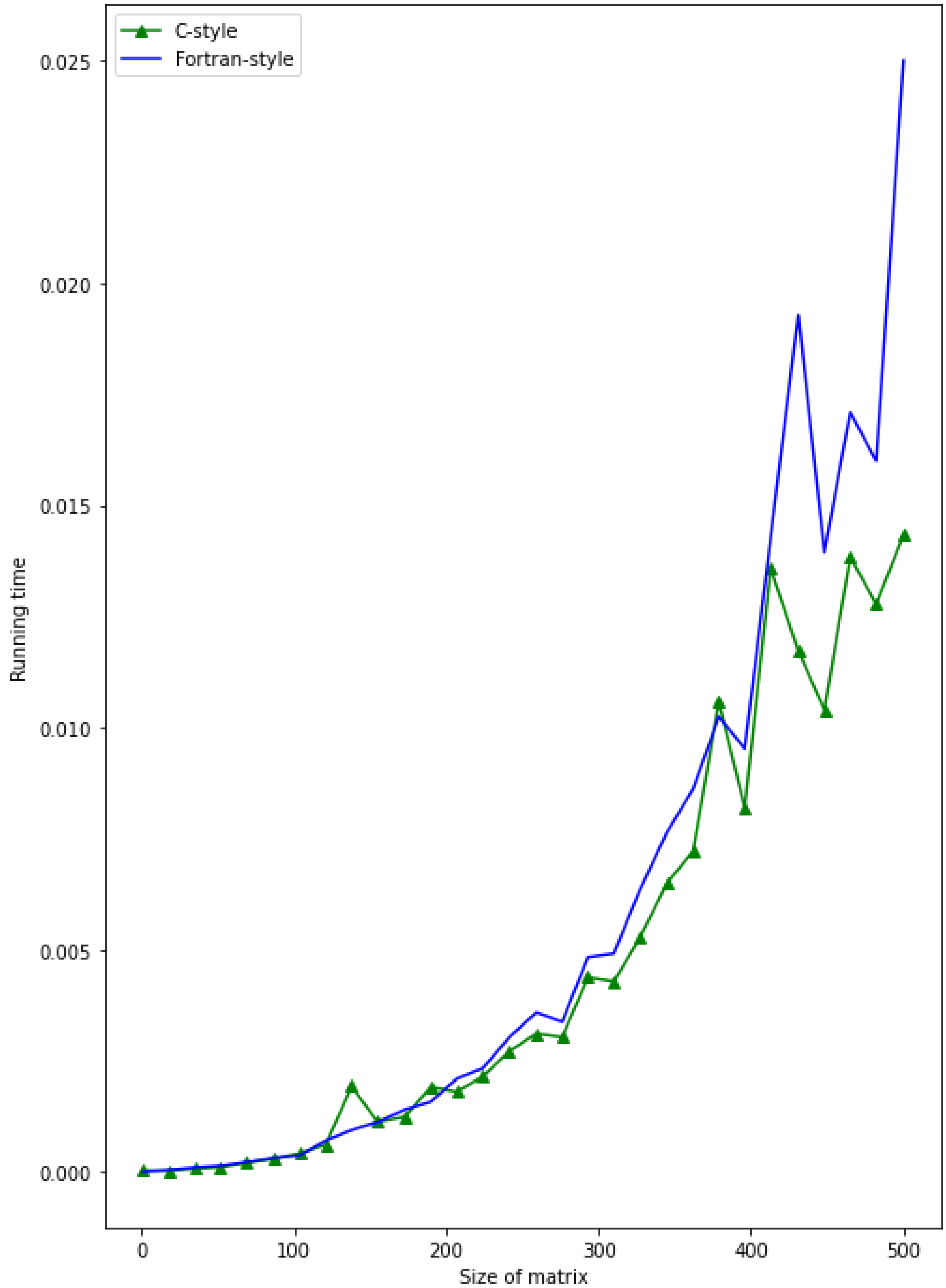


Comparison of C-style and Fortran-style ordering

```
def C_time_counter(mat1, mat2):
    faster_matrix_product_copy(mat1, mat2)


def Fortran_time_counter(mat1, mat2):
    faster_matrix_product_copy(mat1, mat2)


x = np.linspace(1, 500, 30, dtype=np.int32)
y_C = []
y_Fortran = []
for i in x:
```

```
        mat_1 = np.random.rand(i, i)
        mat_2 = np.random.rand(i, i)
        y_C.append(timeit(lambda: C_time_counter(mat_1, mat_2), number=1))
        y_Fortran.append(timeit(lambda: Fortran_time_counter(
                np.asfortranarray(mat_1), np.asfortranarray(mat_2)),
                number=1))


plt.figure(figsize=(8,12))
plt.plot(x, y_C, "g^-")
plt.plot(x, y_Fortran, "b-")

plt.xlabel("Size of matrix")
plt.ylabel("Running time")
plt.legend(["C-style", "Fortran-style"])
```

```
<matplotlib.legend.Legend at 0x7ff467aeb850>
```



The C-style ordering is faster than Fortran-style ordering, because in matrix multiplication operations, the program will read rows of data, and it is more efficient if the read contents are stored in adjacent memory addresses. In C-style ordering, data from the same row is stored in adjacent addresses, but in Fortran-style ordering, data from the same row is stored in separate memory.