NumPy vs. Pure Python

How NumPy beats Python in speed and efficiency for data processing and math operations

Hernández Pérez Erick Fernando

NumPy

```
import numpy as np
```

Figure 1: NumPy

NumPy is one of the core libraries for numerical computation in Python, designed to handle large volumes of data with remarkable efficiency. Its primary advantage lies in its use of multidimensional arrays (ndarrays), which enable mathematical and statistical operations to be executed significantly faster than with conventional Python lists. This performance boost comes from NumPy's underlying C implementation, which optimizes memory access and computation speed.

Beyond its array-handling capabilities, NumPy provides a comprehensive suite of high-level mathematical functions, including support for linear algebra, Fourier transforms, and random number generation. Additionally, it facilitates seamless integration with C, C++, and Fortran, making it an essential tool for scientific computing and data-intensive applications in Python.

Why NumPy?

Given Python's built-in capabilities, one might wonder: why use NumPy instead of native lists for numerical computations? The answer lies in efficiency, memory optimization, and ease of manipulation of large datasets.

Before illustrating the advantages, it is better to illustrate the shortcomings; or rather, the areas of opportunity that pure Python offers us.

- **Memory Overhead**: Lists in Python store references to objects rather than raw numerical data, which increases memory consumption.
- **Performance**: Since Python lists are dynamically typed and interpreted, operations on large datasets tend to be significantly slower.
- **Lack of Vectorization**: Python's lists do not support element-wise operations directly, requiring explicit loops that increase execution time.

It certainly seems like a bleak picture, but once we understand this we can get into what matters most to us, which are the benefits of NumPy.

- Efficient data storage.
  - **Contiguous arrays in memory**: NumPy stores data in adjacent blocks of memory, allowing faster access compared to Python lists, whose elements can be scattered.

  - **Homogeneous data types**: All elements in a NumPy array are of the same type, facilitating hardware-level optimizations and the use of SIMD (Single Instruction, Multiple Data) instructions for simultaneous calculations.
- Vectorized operations.
  - **Avoids explicit loops**: NumPy allows operations to be performed on entire arrays without the need for loops in Python, reducing interpretation overhead.
  - **Broadcasting**: Facilitates operations between arrays in different ways without the need to create intermediate copies, optimizing memory and performance.
- Use of optimized libraries.
  - NumPy takes advantage of BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package), highly optimized libraries in C and Fortran to perform mathematical operations with maximum efficiency.
- Lower interpretation overhead.
  - While Python executes code line by line through its interpreter, NumPy delegates many operations to compiled code, significantly speeding up its execution.

Putting it to the test

The performance of matrix multiplication using Python lists will be compared against the optimized implementation using NumPy. To do this, the necessary libraries are first imported: matplotlib.pyplot, numpy, pandas, seaborn, random and time. Then, an empty DataFrame with two columns, time_python and time_numpy, will be created in which the execution times of both methodologies will be stored. The dimensions of matrices A and B, both of size 200x200, are defined and a number of iterations of 200 will be set to ensure a statistically significant comparison.

During each iteration, the execution time of the matrix multiplication using Python lists will be measured first. Two matrices with random values between 0 and 1 are generated, using random.random(), and the multiplication will be performed using list comprehension and the zip() function. Once the resulting matrix is obtained, the elapsed time will be stored in the variable time_python. The same operation will then be performed using NumPy. To do this, two random matrices will be generated using np.random.random() and multiplied using np.matmul(). The execution time will be stored in time_numpy.

Once the times for each iteration have been obtained, they will be recorded in the DataFrame, along with cumulative values of the execution time for each approach. The total time spent on each iteration will be calculated and the speedup will be determined, which indicates how many times faster NumPy is than the Python list implementation.

The test

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import random
import time

df = pd.DataFrame([],columns=['time_python', 'time_numpy'])

rows_A, cols_A = 200, 200
rows_B, cols_B = 200, 200
iteration = 200

for _ in range(iteration):
    start = time.perf_counter()
    A = [[random.random() for _ in range(cols_A)] for _ in range(rows_A)]
    B = [[random.random() for _ in range(cols_B)] for _ in range(rows_B)]
    result = [[sum(a*b for a, b in zip(A_row, B_col)) for B_col in zip(*B)] for A_row in
    ↪  A]
    end = time.perf_counter()

    time_python = end - start

    start = time.perf_counter()
    A_np = np.random.random((rows_A, cols_A))
    B_np = np.random.random((rows_B, cols_B))
    result_numpy = np.matmul(A_np, B_np)
    end = time.perf_counter()

    time_np = end - start

    df.loc[len(df)] = [time_python, time_np]

df['cumsum_time_python'] = df['time_python'].cumsum()
df['cumsum_time_numpy'] = df['time_numpy'].cumsum()
df['total'] = df["time_python"] + df["time_numpy"]
df["speedup"] = df["time_python"] / df["time_numpy"]
print(df['total'].sum())
df.index = range(1, len(df) + 1)
```

	time_python	**time_numpy**	**speedup**
count	200.000000	200.000000	200.000000
mean	0.827737	0.002263	431.216959
std	0.227549	0.001592	126.325025
min	0.633378	0.001459	103.101393
25%	0.708646	0.001528	363.210407
50%	0.731823	0.001564	461.170413
75%	0.782821	0.002166	486.179823
max	1.807464	0.008900	881.579600

Table 1: Descriptive Statistics for Time and Speedup

Visualization

```
fig, axes = plt.subplots(3,2,figsize=(13,17))

sns.histplot(df,x=df['time_python'],kde=True, kde_kws=dict(cut=3), ax=axes[0,0])
sns.histplot(df,x=df['time_numpy'],kde=True, kde_kws=dict(cut=3), ax=axes[0,1])

axes[1,0].plot(df['time_python'], label='Python')
axes[1,0].plot(df['time_numpy'], label='Numpy')
axes[1,0].set_title("Comparison of pure Python vs NumPy")
axes[1,0].set_xlabel("Iterations")
axes[1,0].set_ylabel("Time (seconds)")
axes[1,0].set_xticks(x)
axes[1,0].legend()

axes[1,1].fill_between(range(1,len(df) + 1),df['cumsum_time_python'], alpha=0.5,
↪  label='Python')
axes[1,1].plot(df['cumsum_time_python'])
axes[1,1].fill_between(range(1,len(df) + 1),10*df['cumsum_time_numpy'], alpha=0.5,
↪  label='10 * Numpy')
axes[1,1].plot(10*df['cumsum_time_numpy'])
axes[1,1].set_title("Cumulative times of pure Python vs 10*NumPy")
axes[1,1].set_xlabel("Iterations")
axes[1,1].set_ylabel("Time (seconds)")
axes[1,1].set(xlim = (min(x), max(x)), xticks = x)
axes[1,1].legend()

axes[2,0].plot(df['speedup'], color='green', label='Improvement')
axes[2,0].set_title("NumPy performance improvement")
axes[2,0].set_xlabel("Iterations")
axes[2,0].set_ylabel("Performance improvement")
axes[2,0].set(xticks = x)
axes[2,0].legend()

sns.violinplot(df['speedup'], ax=axes[2,1])
axes[2,1].set_ylabel('Performance improvement')
axes[2,1].set_title('Violin plot Performance improvement')

fig.suptitle('Comparison of times and performance', fontsize=16, y=0.90)
```

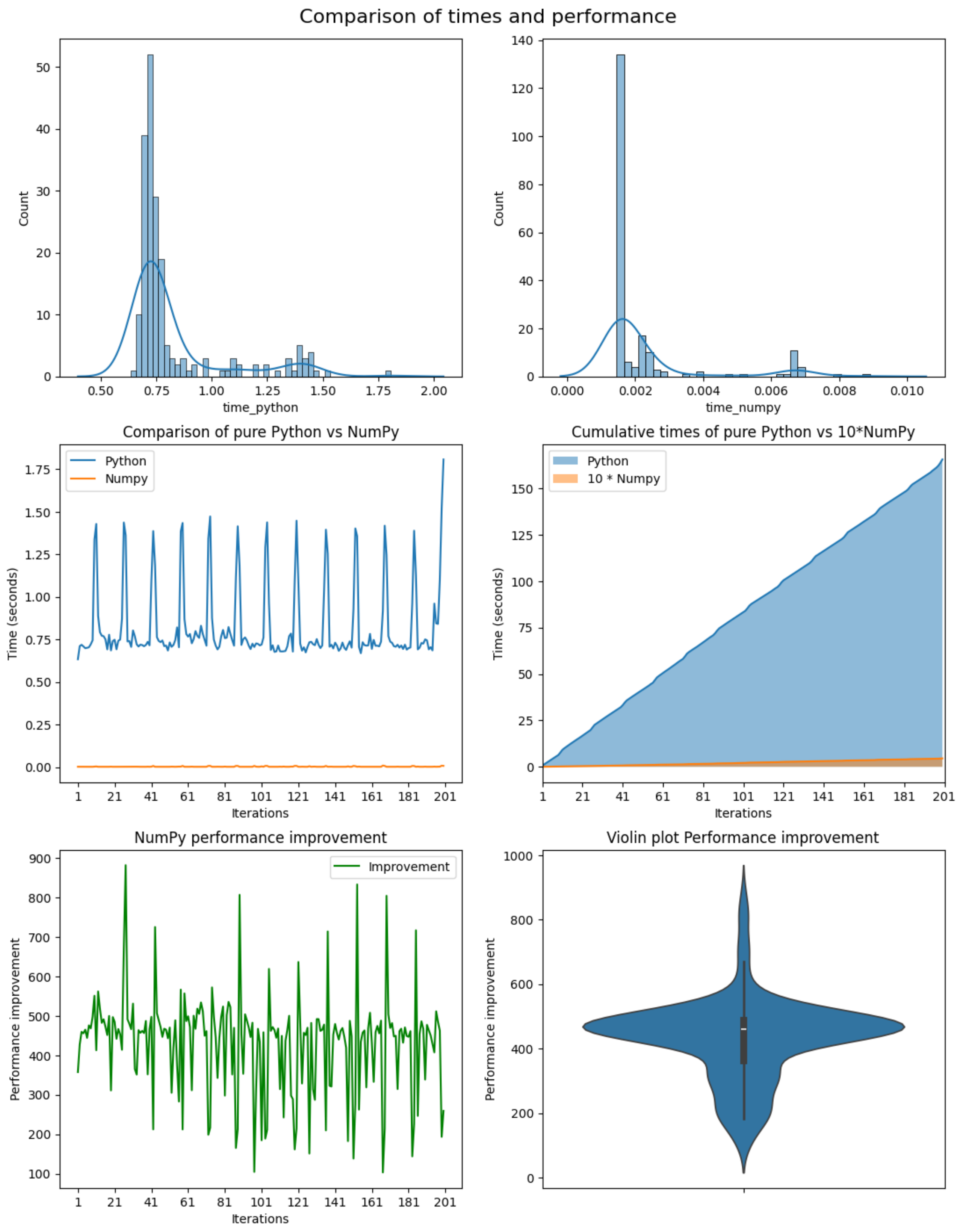

Figure 2: Visualization

Analysis of the data shows a significant difference in performance between operations performed with pure Python and those performed with NumPy for matrix multiplication. As for the execution time in Python (time_python), the average time is observed to be 0.8277 seconds, reflecting the less efficient nature of Python for performing numerical calculations compared to specialized solutions. However, the variability of the times is notable, with a standard deviation of 0.2275 seconds, indicating that, although the average time is reasonably stable, there are certain iterations that experience much higher times, reaching a maximum value of 1.8075 seconds. This could be related to factors such as Python's garbage collector or fluctuations in processor usage.

On the other hand, execution times with NumPy (time_numpy) are considerably lower, with an average time of 0.0023 seconds, highlighting the optimization that NumPy offers for array manipulation and performing mathematical operations. Furthermore, the standard deviation of 0.0016 seconds shows that NumPy execution times are much more consistent and stable compared to pure Python, ranging from a low of 0.0015 seconds to a high of 0.0089 seconds, indicating that even in the worst cases, NumPy is still considerably faster than Python operations.

The average speedup of 431.2170, which reflects how many times faster NumPy is compared to Python, is one of the most notable metrics. This value indicates that, on average, NumPy operations are more than 400 times faster than those performed with lists in Python. Although this speedup is impressive, variability is also present, with a standard deviation of 126.3250, suggesting that performance can vary depending on the iterations. In some cases, the speedup can be even higher, reaching up to 881.5796 times, showing that in certain iterations NumPy can be extremely efficient. However, it is also observed that even in the lower speedup cases, NumPy's performance is still much faster than Python's, with a minimum of 103.1014 times faster.

In summary, the data confirms that NumPy is a noticeably more efficient tool than Python for linear algebra and numerical mathematics operations. The average speedup suggests an improvement of more than 400 times, and the stability of the times with NumPy versus the greater variability in Python highlights the advantage of using specialized libraries such as NumPy for computationally intensive tasks.

Analysis of the data shows a significant difference in performance between operations performed with pure Python and those performed with NumPy for matrix multiplication. As for the execution time in Python (time_python), the average time is observed to be 0.8277 seconds, reflecting the less efficient nature of Python for performing numerical calculations compared to specialized solutions. However, the variability of the times is notable, with a standard deviation of 0.2275 seconds, indicating that, although the average time is reasonably stable, there are certain iterations that experience much higher times, reaching a maximum value of 1.8075 seconds. This could be related to factors such as Python's garbage collector or fluctuations in processor usage.

On the other hand, execution times with NumPy (time_numpy) are considerably lower, with an average time of 0.0023 seconds, highlighting the optimization that NumPy offers for array manipulation and performing mathematical operations. Furthermore, the standard deviation of 0.0016 seconds shows that NumPy execution times are much more consistent and stable compared to pure Python, ranging from a low of 0.0015 seconds to a high of 0.0089 seconds, indicating that even in the worst cases, NumPy is still considerably faster than Python operations.

The average speedup of 431.2170, which reflects how many times faster NumPy is compared to Python, is one of the most notable metrics. This value indicates that, on average, NumPy operations are more than 400 times faster than those performed with lists in Python. Although this speedup is impressive, variability is also present, with a standard deviation of 126.3250, suggesting that performance can vary depending on the iterations. In some cases, the speedup can be even higher, reaching up to 881.5796 times, which shows that in certain iterations NumPy can be extremely efficient. However, it is also observed that even in the lower speedup cases, NumPy's performance is still much faster than Python's, with a minimum of 103.1014 times faster.

In summary, the data confirms that NumPy is a noticeably more efficient tool than Python for linear algebra and numerical mathematics operations. The average speedup suggests an improvement of more than 400 times, and the stability of the times with NumPy versus the greater variability in Python highlights the advantage of using specialized libraries like NumPy for computationally intensive tasks.

And that's not even mentioning memory...

References

[1] Brownlee, J. (2023). *What is BLAS and LAPACK in NumPy.* SuperFast Python. https://superfastpython.com/what-is-blas-and-lapack-in-numpy/

[2] NumPy. (n.d.). *NumPy v1.24 Manual.* https://numpy.org/doc/stable/index.html