

Coding Example: Blue Noise

Sampling

Introduction

This lesson gives a brief introduction to code vectorization and explains it with the help of an example.

Problem vectorization is much harder than code vectorization because it means that you fundamentally have to rethink your problem in order to make it vectorizable. Most of the time this means you have to use a different algorithm to solve your problem or even worse... to invent a new one. The difficulty is thus to think out-of-the-box.

To illustrate this, let's consider a simple problem where given two vectors \mathbf{X} and \mathbf{Y} , we want to compute the sum of $\mathbf{X}[\mathbf{i}]^*\mathbf{Y}[\mathbf{j}]$ for all pairs of indices \mathbf{i} , \mathbf{j} .

One simple and obvious solution is given below.

```
def compute_python(X, Y):
    result = 0
    for i in range(len(X)):
        for j in range(len(Y)):
            result += X[i] * Y[j]
        return result

print(compute_python([1,2],[1,2]))

RUN

SAVE RESET ()
```

However, this first and naïve implementation requires two loops and we already know it will be slow:

```
import numpy as np
main.py
                                  from tools import timeit
tools.py
                                  def compute_python(X, Y):
                                    result = 0
                                    for i in range(len(X)):
                                      for j in range(len(Y)):
                                        result += X[i] * Y[j]
                                        return result
                              11 X = np.arange(1000)
                              12 timeit("compute_python(X,X)", globals())
    RUN
                                                                                       SAVE
                                                                                                   RESET
                                                                                                              0
                                                                                                             ×
                                                                                                        0.7715
Output
 10000 loops, best of 3: 2.11 usec per loop
```

How to vectorize the problem then? If you remember your linear algebra course, you may have identified the expression <code>X[i] * Y[j]</code> to be very similar to a matrix product expression. So maybe we could benefit from some NumPy speedup. One wrong solution would be to write:

```
def compute_numpy_wrong(X, Y):
return (X*Y).sum()

print(compute_numpy_wrong([1,2],[1,2]))
```

This is wrong because the X*Y expression will actually compute a new vector Z such that Z[i] = X[i] * Y[i] and this is not what we want. Instead, we can exploit NumPy broadcasting by first reshaping the two vectors and then multiplying them:

```
1 def compute_numpy(X, Y):
2     Z = X.reshape(len(X),1) * Y.reshape(1,len(Y))
3     return Z.sum()
```

expected result. Let's see how much speed we gain in the process:

Here we have Z[i,j] == X[i,0]*Y[0,j] and if we take the sum over each elements of Z, we get the

```
1 import numpy as np
main.py
                              2 from tools import timeit
                              3 def compute_numpy(X, Y):
tools.py
                                  Z = X.reshape(len(X),1) * Y.reshape(1,len(Y))
                                  return Z.sum()
                                 X = np.arange(1000)
                              8 timeit("compute_numpy(X,X)", globals())
    RUN
                                                                                                           03
                                                                                     SAVE
                                                                                                  RESET
                                                                                                           ×
Output
                                                                                                     0.7215
10 loops, best of 3: 2.21 msec per loop
```

If you look again and more closely at the pure Python version, you can see that the inner loop is using

This is better, we gained a factor of ~150. But we can do much better.

X[i] that does not depend on the j index, meaning it can be removed from the inner loop. Code can be rewritten as:

But since the inner loop does not depend on the i index, we might as well compute it only once:

Using the same approach, we can now write:

1 def compute_numpy_better_3(x, y):

from the np.sum function and write:

1 def compute numpy better(x, y):

Finally, having realized we only need the product of the sum over X and Y respectively, we can benefit

```
It is shorter, clearer and much, much faster:
```

1 import numpy as np
main.py 2 from tools import timeit

return np.sum(y) * np.sum(x)

def compute_numpy_better_2(X, Y):

Code vectorization

The latter is the most difficult but the most important because this is where you can expect huge gains in speed. In this simple example, we gain a factor of 150 with code vectorization but we gained a factor of

Problem vectorization

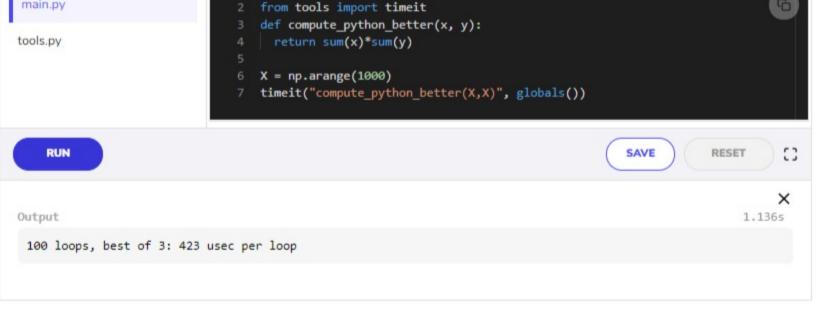
70,000 with problem vectorization, just by writing our problem differently (even though you cannot expect such a huge speedup in all situations).

However, code vectorization remains an important factor, and if we rewrite the last solution the Python way, the improvement is good but not as much as in the NumPy version:

1 def compute_python_better(x, y):
2 return sum(x)*sum(y)



main.py 1 import numpy as np
2 from tools import timeit





Next, we will discuss a case study and see how we can solve it using the problem vectorization approach.