Numba

Gabriel S. Cabrera

August 15, 2018

Contents

1 Introduction

`Numba` is a `Python` optimization package that converts `Python` functions into machine code. It can be used in situations where `NumPy` is not optimal, such as in the integration of second order differential equations; one example would be the equation for a simple oscillating spring.

1.1 Example: Spring Oscillator

$$m\frac{d^2}{dt^2}x(t) = -kx(t) \tag{1}$$

(1) can be numerically evaluated for a set of initial conditions using Euler-Cromer Integration:

Algorithm 1 An Euler-Cromer integration algorithm for a simple spring oscillator

t = 0
while $(t \leq T)$ **do**
    $a_i = -\frac{k}{m}x_{i-1}$
    $v_i = v_{i-1} + a_i\Delta t$
    $x_i = x_{i-1} + v_i\Delta t$
    $t = t + \Delta t$
end while

This can be written as a `Python` function:

```python
def integrate(k, m, x0, v0, T, dt):
    N = int(T//dt)
    t = [0]
    x = [x0]
    v = [v0]
    for n in range(N):
        a = -k*x[-1]/m
        v.append(v[-1] + a*dt)
        x.append(x[-1] + v[-1]*dt)
        t.append(t[-1]+dt)
    return t, x, v
```

If we set our parameters and initial conditions to $k = 0.1\text{N/m}$, $m = 1 \times 10^5\text{kg}$, $x_0 = 0\text{m}$, $v_0 = 2\text{m/s}$, $T = 2 \times 10^5\text{s}$, and $\Delta t = 0.01\text{s}$, it takes `Python` approximately 40 seconds[1] to run this loop; the result is shown in Figure 1.

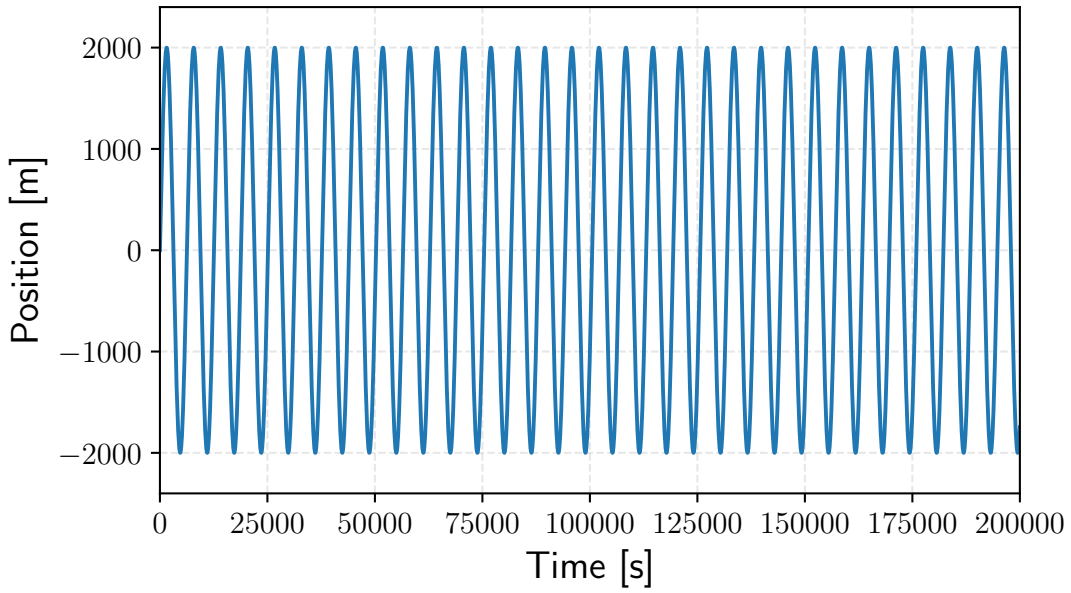

Figure 1: The position of a one-dimensional oscillating spring over time

Before we continue, note that we cannot vectorize this function completely since the acceleration at a given step depends on the position of the previous step; this means that `NumPy` is not particularly useful to us at the moment – we can however implement `Numba` compilation via the addition of only two lines of code:

```python
from numba import jit

@jit
def integrate(k, m, x0, v0, T, dt):
    N = int(T//dt)
    t = [0]
    x = [x0]
```

[1]The time it takes to run this program depends highly on the computer in question but bear with us in the meantime, as we are most interested in the percent increase in speed via implementing `Numba`.

```
    v = [v0]
    for n in range(N):
        a = -k*x[-1]/m
        v.append(v[-1] + a*dt)
        x.append(x[-1] + v[-1]*dt)
        t.append(t[-1]+dt)
    return t, x, v
```

This gives us the exact same result, except this time it ran *significantly* faster – this only took 14 seconds to run when tested on the same hardware as the previous script, a 285% improvement in efficiency!

2 Usage

To implement just-in-time compilation, we will need to import the `jit` function from `Numba`. To use `jit`, we simply insert a decorator just above the desired function as follows:

```
from numba import jit

@jit                    #The decorator
def foo():              #An arbitrary function
    return None
```

To take full advantage of the potential performance boost however, we should use the `nopython` argument; this is declared as follows:

```
from numba import jit

@jit(nopython = True)   #The decorator in nopython mode
def foo():              #An arbitrary function
    return None
```

There are two main modes in which `@jit` can be run – `object` mode and `nopython` mode. In short, `object` mode is:

- More versatile than `nopython` mode – anything that works in standard `Python` will work in a `@jit` optimized function in `object` mode.
- Less than or equally efficient to `nopython` mode, depending on the function contents.
- Activated when setting `nopython = False`, or simply using `@jit` on its own without arguments.

While `nopython` mode is:

- More restrictive as to what the `@jit` function may contain – errors will be raised if `Numba`'s requirements aren't met!
- The most efficient way to run a program, assuming compatibility is possible.
- Activated when setting `nopython = True`.

When omitting `nopython`, `Numba` will be able to compile any given `Python` function – this is called `object` mode and is the most versatile. There is a downside to `object` mode however: in many cases, it will be slower than the alternative `nopython` mode. Generally speaking it is therefore best to use the `nopython` mode, though it comes at the cost of a lack of versatility.

The mechanism at work here is as follows: if we do not restrict our function to `nopython = True`, `Numba` will still try its best to compile as much of the function in `nopython` mode as possible, but will revert to standard `Python` code whenever it is unable to compile a part of the function.

On the other hand, using `nopython` mode prevents `Numba` from reverting to `Python` and will instead raise errors when a portion of the code cannot be compiled, ensuring that a developer can make their program as efficient as possible.

Unfortunately, learning how to debug in `nopython` mode takes time and effort, so here are a few guidelines:

- `Numba` supports `NumPy` arrays, but its support of `NumPy` functions is somewhat limited. [2]
- It is not possible to redefine variable types in `Numba`; the following function would raise an error for example:

```
@jit(nopython = True)
def foo():
    a = 20                  #Declared a variable of type int
    a = 'Hello World'       #Illegal attempt to redefine int <a> to type str
```

- Make sure the function works without the `@jit` decorator in the first place! You may be surprised to find that an error raised by `Numba` is due to an error in your `Python` code itself. For example, the following code snippet would raise a `TypeError` in standard `Python`, but will instead raise a `Numba` error due to the presence of the `@jit` decorator

```
@jit(nopython = True)
def foo():
    a = 1/'hello'           #Raises a Numba error, though it is a Python3 error
```

- Do not pass functions as arguments, this is not allowed. If you wish to access an external function from inside an `@jit` function, it must be called from within the function. It is recommended that the called function also be an `@jit` function.

  The following shows what *not* to do:

```
@jit(nopython = True)
def foo():
    return None

@jit(nopython = True)
def foo2(foo):
    print(foo())
```

  Instead, the following is the correct syntax:

[2] An overview of what is supported can be found here: https://numba.pydata.org/numba-doc/dev/reference/numpysupported.html

```
@jit(nopython = True)
def foo():
    return None

@jit(nopython = True)
def foo2():
    print(foo())
```

There is a lot more to learn in this regard, and you will undoubtedly come across some error messages that appear hieroglyphic in nature. In most cases, a quick trip to StackExchange will suffice, but you may also wish to consult the official documentation for your Numba version: https://numba.pydata.org/doc.html

3 Rates of Efficiency

Although `Numba` allows for the possibility of significant improvements in performance, it is important to realize that this strength manifests itself in cases where a program is running *many* iterations of an `@jit` function. As a result, it is possible that `Numba` may not be all that useful when running a program with very few iterations.

To better understand the relation between iterations and performance improvements we can create two functions with the same exact contents – the only difference between them will be that one has implemented `@jit`, and the other one has not:

```
def total(N):

    '''
        Adds all integers from 0 to N and returns the resulting total sum. Also
        performs a couple more operations to make the calculations heavier.
    '''

    val = 0
    for i in range(N):
        val += i
        val /= i+1
        val *= i+1
    return val

@jit
def jit_total(N):

    '''
        Adds all integers from 0 to N and returns the resulting total sum,
        using <jit>.  Also performs a couple more operations to make the
        calculations heavier.
    '''

    val = 0
    for i in range(N):
        val += i
        val /= i+1
        val *= i+1
    return val
```

We are interested in comparing performance as a function of iterations, so by running these functions for a variety of `N`, timing their runtimes, and calculating their ratios, we see that the full power of `Numba` is not attained until a large number of iterations has occured. – this is shown in Figure 2:

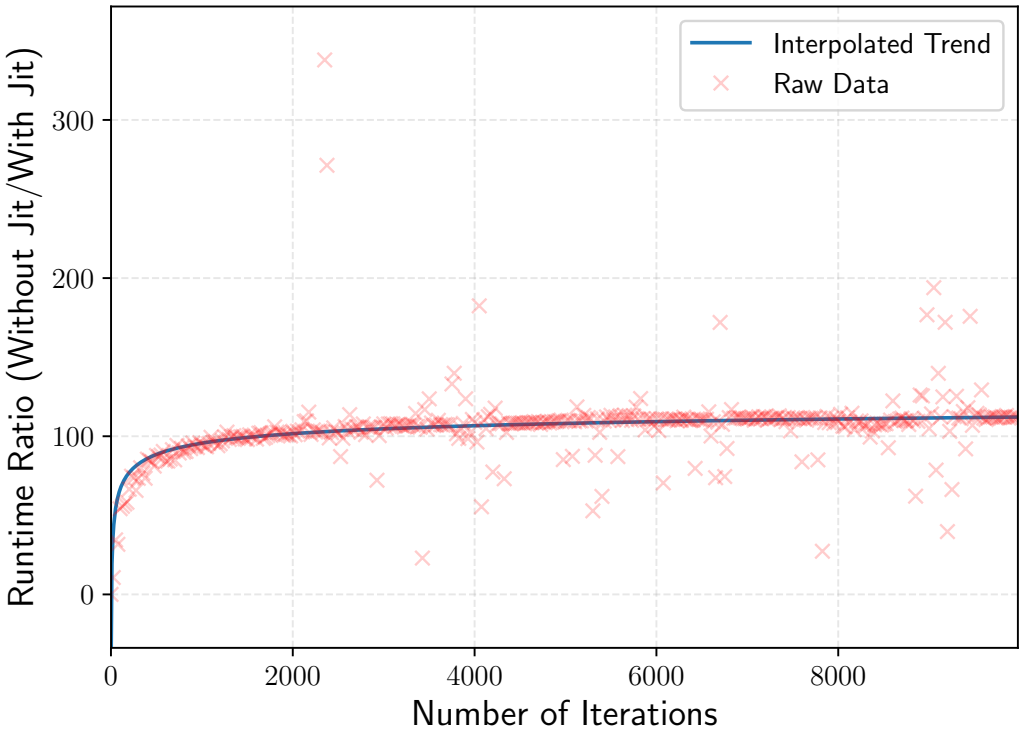

Figure 2: How `Numba` can affect performance based on the number of iterations taken

We can therefore conclude that using `@jit` for a small number of iterations does not give us a signification advantage in performance – however, as we increase the number of iterations, we get an increase in speed that converges to an approximate 100X speed increase[3], meaning that `@jit` is extremely useful when attempting to increase the speed of a loop, but only when the number of iterations is high enough.

[3]Results will vary depending on the function in question, of course.