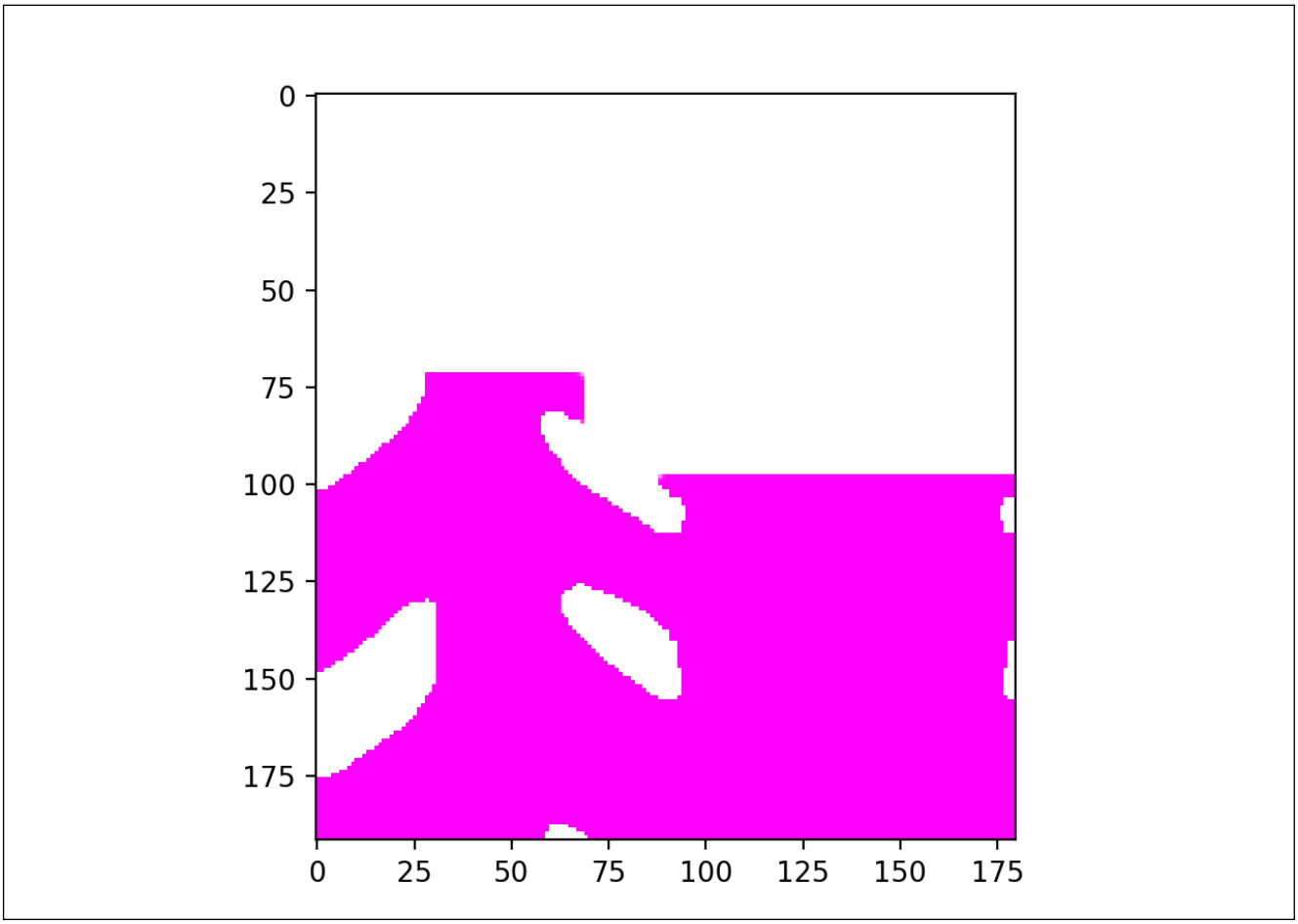
Robot simple test project

Optimization stage 3

J. YANG Zhenyu - February 24, 2020



Introduction

The semester cannot start in Mars due to virus affection. So the work continues at home in rather rough conditions.

As mentioned in previous reports, np.roll() is a function used extensively in the program but not implemented in Numba library. The performance of Numba was tested in winter report stage 2 and was very promising even without GPU acceleration. So it is necessary to write this function by myself. This is the priority in this report.

In the mean time, a small animation function will also be prepared for visualizing the finding process.

In this report, the neuronal space size is 192 * 18000. (~3×10^6)

Before the work of this report

The resources consumption state (optimum condition) is :

Preparation : 2.9 s

Generation of neuronal space : 236 s

Path finding : 850 s.

After the work of this report,

The resources consumption state (optimum condition) is :

Preparation : 2.9 s

Generation of neuronal space : 236 s

Path finding : 850 s.

Implementation of np.roll()

np.roll() was in fact implemented in Numba without support for *axis* argument. However, this support is essential for our program. As the code is open source and the merge commit of this feature can be viewed directly in Github, one simple work through is study and copy those code and add the support for *axis*.

Github issue #3563 - Support for np.roll

This seems rather simple, I just have to modify the method of calculating the index part in original implementation.

```
1383 + @overload(np.roll)
1384 + def np_roll(a, shift):
1385 +
1386 +     if not isinstance(shift, (types.Integer, types.Boolean)):
1387 +         raise TypingError('shift must be an integer')
1388 +
1389 +     def np_roll_impl(a, shift):
1390 +         arr = np.asarray(a)
1391 +         out = np.empty(arr.shape, dtype=arr.dtype)
1392 +         # empty_like might result in different contiguity vs NumPy
1393 +
1394 +         arr_flat = arr.flat
1395 +         for i in range(arr.size):
1396 +             idx = (i + shift) % arr.size
1397 +             out.flat[idx] = arr_flat[i]
1398 +
1399 +         return out
1400 +
1401 +     if isinstance(a, (types.Number, types.Boolean)):
1402 +         return lambda a, shift: np.asarray(a)
1403 +     else:
1404 +         return np_roll_impl
1405 +
```

Numba code of np.roll()

Let A a matrix of dimension n, B the matrix after shifting $shift$ along axis $axis$. Then we'll have for all $A[(x_i)_{i<n}]$, $B[((x_i + (i == shift) * shift) \,\%\, shape[axis])_{i<n}] = A[(x_i)_{i<n}]$.

In order to prevent if or for overlap which consumes considerable amount of resources, a method using modulus operator is proposed.

Let $a = \prod_{j=1}^{n-axis} shape[n-j]$, $b = \prod_{j=1}^{n-axis-1} shape[n-j]$. Apparently $b = a/shape[axis]$

The code is not aware of the dimension of the matrix, so we'll have to operate through flatten array (of dimension 1). A coordinate conversion formula is shown below :

If $A[(x_i)_{i<n}] = A.flat[idx]$, then $idx = \sum_{i=0}^{n-1} x_i \prod_{j=1}^{n-i-1} shape[n-j]$.

So with $B[((x_i + (i == shift) * shift) \,\%\, shape[axis])_{i<n}] = A[(x_i)_{i<n}]$,

we'll have :

$$idx_B = idx_A + ((x_{axis} + shift) \,\%\, shape[axis] - x_{axis}) \prod_{j=1}^{n-axis-1} shape[n-j]$$

Now we've only to get x_{axis} from idx_A, firstly by $\%\, a$, we'll get $\sum_{i=0}^{axis} x_i \prod_{j=1}^{n-i-1} shape[n-j]$.

Then divide by b, take only the integer part we'll finally get x_{axis}.

The code of this process is shown below :

This method will require $axis < n$, however shift can exceed the corresponding dimension's size thanks to the modulo operator.

```python
import numpy as np


A = np.asarray(range(24))
A = A.reshape(3, 4, 2)
axis = 1
Ndim = A.ndim
shift = 3

a = 1
for i in range(Ndim - axis):
    j = i + 1
    a *= A.shape[Ndim - j]

b = (int)(a / A.shape[axis])


B = np.empty(A.shape, dtype=np.int16)
for i in range(A.size):
    # Get x_axis
    x_axis = (int)((i % a) / b)
    idx = (i + ((x_axis + shift) % A.shape[axis] - x_axis) * b) % A.size
    B.flat[idx] = A.flat[i]

print(B)
print(np.roll(A, shift, axis=axis))

```

The second step is to integrate my code to the Numba library. Several problems raised due to poor network connection.

Installing numba at local directory was rather straight forward.

After several attempts, my solution is to connect the school VPN, then route my proxy software to use this VPN, then use the proxy to connect *the* Internet. Without these connections I could not even install the library's dependencies.

Results show that my implementation of np.roll() speeds **down** the calculation. It takes from ~2s to 30+s. This indicates that above ideas are not efficient enough. I've checked the implementation of np.roll() in NumPy library. It is not possible to implement in Numba because almost all the intermediate functions are not implemented. It will need tremendous amount of work to implement them all and I'm not sure that I can really do it.

I may have to find another way of vectorization to avoid this. Or I may have to write some C code to make this part run fast enough.

Implementation of animation program

This is pretty straight forward. It is very simple to convert the path to a suite of positions and then to an animation.

I found a problem with previous implementation, I'll need to lock the target area with value 1. Or it will no conserve. It is quite difficult to do with my previous vectorization method, but with the method mentioned below it will be very easy.

However, the video is not really so important so I decide to export a picture with the path.

Plan of hexapod robot

I will firstly try to reproduce last team's work. That is to say, consider only the path planning part. For this part, our innovative idea could be the usage of modulus which will help control energy and time consumption. (In fact, it is a factor in the finding process) As different gait consumes different account of energy. They didn't consider this aspect before.

To design this system, I've drawn another diagram to make everything clear. The space is about 10e7 which might be acceptable.

In each position, the robot is in a gait state. Like moving, climbing or standing still. These gaits will have parameters like a direction, climbing angle. Which result in a 4 dimension matrix which is at maximum 10e7 size. And the result is directly the entire path.

Although our plan is to combine them together, the computational power could not allow such simulations. I'll need to further divide the system. It is not yet feasible. Let us assume that with GPU acceleration and np.roll() implemented with Numba. It will maximumly reduce the time consumption of a 10e8 matrix to a 1s order of magnitude. But for a quadruped robot, the matrix size is at minimum 10e10 for a space less than $1m^3$. This is a method of exhaustion and too many redundant calculations are made.

Conclusion

In this stage of work, I tried to implement np.roll() in Numba to further increase the calculation speed but failed. The algorithm that I designed is elegant to me but not to computers. It takes more time than without Numba.

An animation program is created. Its code could be moved to serve our final project.

As for the real robot part, a new plan of working is proposed. Our original innovation point has difficulties seemingly impossible to overcome. So a new one is mentioned.

Due to various reasons, this stage proceeded quite slowly. I'll have to speed up in next stage to catch up the progress.

Next stage will mainly be about the path planning program designing. Also, another method of vectorization has came to me recently, in which np.roll() will not be needed. I'll also try it.