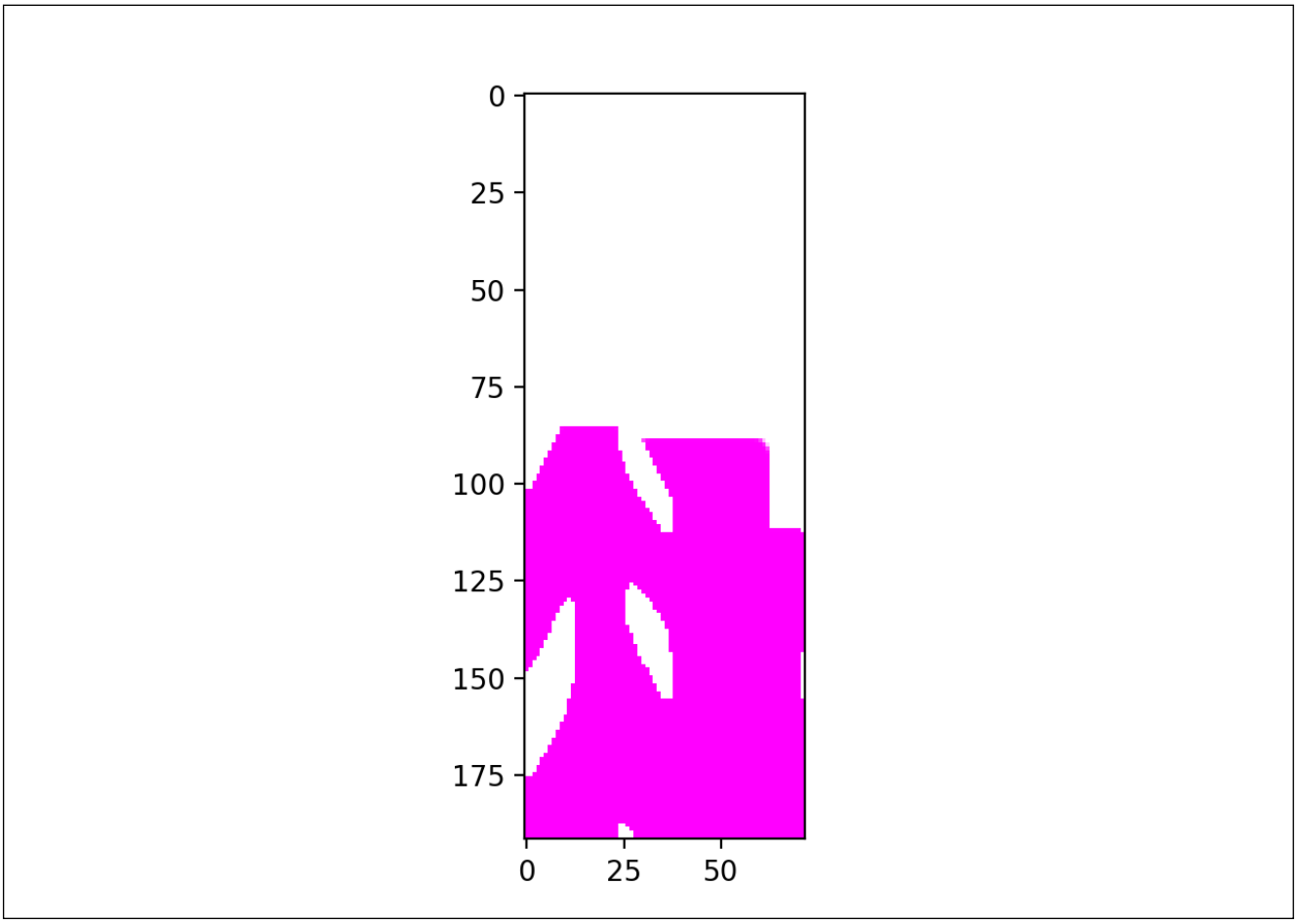
Robot simple test project

Optimization stage 1

J. YANG Zhenyu - February 1, 2020



1, Introduction

This winter holiday provides time that could help finish the simple test project of a two-arm robot in a 2D space and at the same time, prepare information of hexapod robots to continue the real project. This report consists only the various methods that I applied to accelerate the simple 2D - one arm robot.

At first the debug of original simple project before vectorization is carried out. After fixing several errors the program runs without problem. A small 2D neuronal space is generated and then the path is found.

Pre-calculation of robot's coordinates is then implemented, its improvement is to be evaluated. See chapter 2.

As recommended and needed by the Python language for efficient computing, *vectorization* is needed to make the program finish running in an acceptable amount of time. But it's not so easy as the neuronal space's evolvement requires complex calculations that are not provided by native Numpy library. A search through Anaconda of such tools is needed.

In this report, neuronal space size is 192 * 18.

For instance (to the end of this report), the resources consumption state (optimum condition) is :

Preparation : 0.12 s

Generation of neuronal space : 0.35 s

Path finding : 0.085 s

2, Evaluation of the improvements of pre-calculation

Without pre-calculation, generation of neuronal space takes about : 44.5 s.

With pre-calculation, generation of neuronal space takes about : 0.41 s, the preparation of coordinates takes about 0.14 s.

An improvement of approximately 100 × can be expected from this method. Greater the space, more significant will be the improvement.

Most importantly this method lowers the importance of the efficiency of the process of solving robot's presence, which is in addition, not a simple subject.

3, Vectorization

Since some blocks in the neuronal space are locked (obstacle area) O, direct manipulations are inconvenient.

One work through is to assign particular values for these points that won't change while iterating and will affect adjoint points in the same way 0 (blank space) does. Such method is not discovered.

But one method came up to me while writing precedent paragraph. See

Vectorization method 1

This method, however, consumes much more memory than direct calculation, but all the calculations are done with arrays.

A detailed description cannot be typed with words, basically it stores obstacle area in another matrix and reset the value at this points to zero at each iteration. I'll set up LateX documents to explain this. LateX will also be applied for future documents.

As tested, this method is, without any acceleration by GPU or Numba, 300 × faster than original method.

For this method, the resources consumption state is :
Prepration : 0.12 s
Generation of neuronal space : 0.35 s
Path finding : 0.085 s

Alternate method

It might be more efficient to store all the feasible area F in an array, and use a vectorized function to calculate this array.

The major difficulty is that this array contains only the coordinates to another matrix — neuronal space. The process of retrieving info from the matrix based on coordinates stored in the array might not be a good practice and slow.

For this method, the resources consumption state is :
Prepration : 0.12 s
Generation of neuronal space : 0.35 s
Path finding : 25.13 s

4, Acceleration

Major acceleration target is parallel computing using GPU cores. Before that vectorization must be done.

I've gone through several Python tutorials without finding anything alike. Maybe I will have to write this function from the basis. To realize this, I might need a lot more knowledge and experience about the Python language.

Possible related concepts to study :

1. ~~Numpy.vectorize()~~

   The numpy.vectorize() function has functionalities to make self designed functions to take array arguments but it is only for convenience rather than performance. It uses a for loop inside. It will not help. ABORT.

2. Numba

   Numba is a Python compiler that transforms Python code to high performance machine code which supports NumPy arrays and functions and loops. That is to say by using this I can avoid probably the vectorization part of this job. It could also work with CUDA to use Nvidia GPU. I will firstly apply this package to the original finding() which uses excessively loops. And test its performance boost.

   Usage :

   Dependencies : Anaconda 3 and numba package installed.

   Add $@jit(nopython = True)$ before the $findPath()$ function will make the function run with Numba compiler.

   A simple test shows that a pure loop after compilation (that is to say used compiled code) is $400\times$ faster. But that's in optimized condition, nevertheless I might be able to expect a 20 times faster $findPath()$ function if properly configured.

   I should also isolate all the functions instead of using global variables. Better practice without which Numba cannot run.

   Errors encountered :

   1. $space2 = space$ is not supported. This was created to update space. While the next value in the neuronal space requires the previous neuronal space to present in memory this is required. FIX : try move space2 = space outside the funciton. **CANNOT RUN -> 28s**
   2. $checkReach()$ that is inside the $findPath()$ function must also be in the $@jit$. Or it won't work. While $checkReach()$ does very little work, I'm combining it in $findPath()$. **TIME : 28 s**
   3. $np.max(min)$ is different from $max(min)$, this will causes problems. But as Numba supports native $max(min)$ there is no need to bother.
   4. np.zeros not supported in $(nopython = true)$ mode. Trying to fix.

Numba is important, but it didn't work well with un-vectorized function. It will be further studied in next report.

5. CUDA

CUDA, however, will be essential as it's for instance the only way to use GPU with Python code. But unfortunately Mac without Nvidia Graphic drivers cannot run CUDA code. GPU acceleration will be left to after my return to school.

6. ~~Numpy Universal functions - ufunc~~
7. ~~Numpy 2D array indexing~~

After the vectorization method found, there is no need for such thing unless it turns out that that method cannot be done in the robot case.

4, Consideration

Now that this project is already in order, I doubled the size of angles to see the influence.

For this method, the resources consumption state is :

Prepration : 0.344 s

Generation of neuronal space : 1.443 s

Path finding : 0.117 s.

We see that now the most consuming step is generation of neuronal space, as for now it's still using loops to generate. In next stage of work, I will try to solve this.

Also, I'll try to put two-arm system to further test the optimizations done.