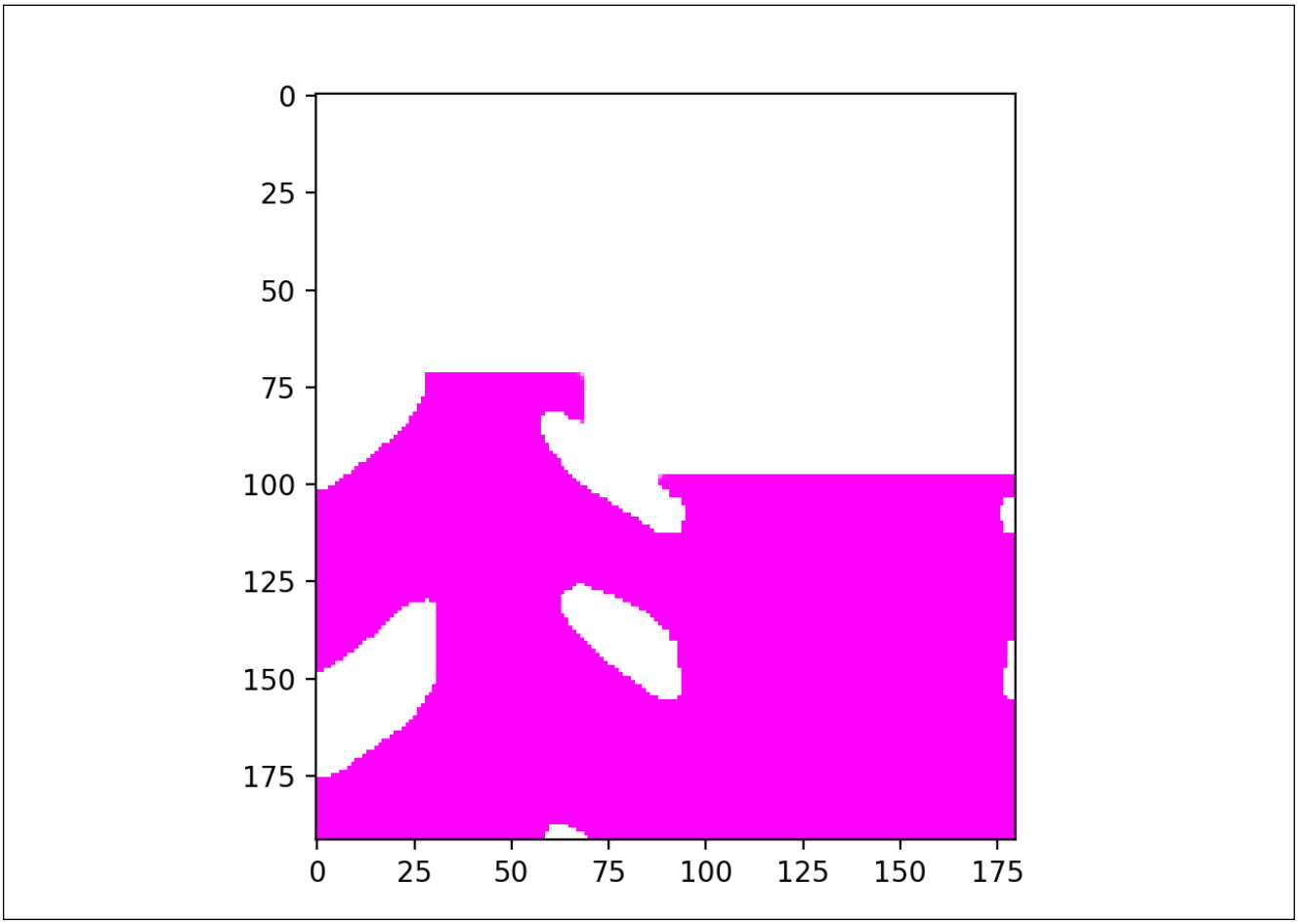
Robot simple test project

Optimization stage 2

J. YANG Zhenyu - February 11, 2020



1, Introduction

This is stage 2 of the Winter holiday work report and simple test project optimization. Comparing to the first stage, neuronal space is ***100 times*** bigger.

Direct calculation without any more optimizations shows that preparation takes 100x more time. Generation of neuronal space takes 1000x more time while path finding takes less than 30x times more time. Vectorization's performance is better than expected.

So in this stage of work, I'll try to find a method to vectorize the generation of neuronal space, and if possible apply Numba to both preparation and generation for that they are all parallel work.

In this report, n will refer to space dimension, m will refer to robot dimension (robot's free degree). For simple test n = 1.5 (cannot move in y axis) and m = 1. For real hexapod robot, n = 3 m = 18.

In this report, neuronal space size is 192 * 1800.

Before this report, the resources consumption state (optimum condition) is :

Preparation : 7.7 s

Generation of neuronal space : 380 s

Path finding : 3.2 s

After, the resources consumption state (optimum condition) is :

Preparation : 0.1 s

Generation of neuronal space : 24 s

Path finding : 2.4 s

Study of related concepts

1, NumPy universal functions - ufunc

Traditional ufunc is inconvenient and requires writing C code. But Numba provides a way around this which is vectorize() decorator. http://numba.pydata.org/numba-doc/latest/user/vectorize.html

This is trivial with vectorize() decorator.

2, Numba

This time Numba will be studied more throughly by reading the user manual in detail.

1.4.2 Calling and inlining other functions

Numba is able to call in a $@jit$ function another $@jit$, it's only you will have to $@jit$ all of them.

Numba doesn't care about global variables.

I should also avoid global variables, considering rearchitect the code to use only arguments in functions.

NumPy $np.ones()$ ***function cannot run under*** $@jit$

Real problem is that I should not use $np.ones$([192,180]) but $np.ones$((192,180)). This is better practice although they do exactly same things. A problem is that it's not supporting dtype. CORRECT : it's not supporting standard int but we can use np.int64.

NumPy $np.roll()$ ***not supported when axis argument presents***

This is a real problem, by default the matrix is flattened before shifting. That is not what I want. After digging the issues in Github directory of Numba, I found that this is a very well known but not solved problem.

It is, however, recommended that I use for loop to replace roll because Numba could speed up it.

For loops without Numba **> 1000 s.**

For loops with Numba **28 s**

It is not good practice, **I may have to write this function in Numba by myself.**

Use of type 'reflected list' is scheduled for deprecation.

Although, it's just a warning, but I think it's affecting the performance. Reflected list is a concept of Python, in this case, it's referring to the list of list I'm using — *presences*. It's used for storage of robot's configuration to prevent run time calculations. A simple work through is to *turn it to from a list of 2 dimension matrix to a 3 dimension matrix*. A[1, :, :] can extract

directly a 2D matrix from 3D matrix. **Tested, this method works fine.** Though the generation process is slightly slowed down.

Vectorization of generation process

Generation process takes majorly three parts, first is prepare the presences, then calculate the neuronal space and lastly find the target area and mark them in the neuronal space. All loops inside all of them are internally irrelevant, so it's completely ok to calculate in parallel.

Idea 1

Use one number to store value of m dimensions by modulo operator. Store coordinates in a m dimension matrix and operate on this matrix. When given to Feasible() function, it extracts the values by modulo operator (m-1 times). Thus get the configure of robot and extract its presence using this. Then do a matrix multiplication to check overlap and the result. To achieve this we need to write Feasible() function as a NumPy universal function. (Use vectorize() decorator is fine.)

For the real robot space, I don't think this would be a good idea as not all points are needed. This method cannot avoid those unnecessary calculations. So I reckon that this is not a promising method but I shall try this if GPU acceleration cannot reduce the time consumption of 192 * 18000 to less than 10 s.

Conclusion

In this report, I've accelerated the program with Numba library. Its performance is quite impressive. A vectorization method is proposed but not implemented due to various reasons. Without GPU acceleration (Numba will be able to do that directly with current code in a compatible device) the code is already speeded up by more than 10 times.

But the real robot space is estimated to at least bigger than 10^6. If we increase the neuronal space size to that order of magnitude. Result shows that for really large spaces, finding process is still the most time consuming part.

For neuronal space size is 192 * 18000. (~3×10^6)

The resources consumption state (optimum condition) is :

Preparation : 2.9 s

Generation of neuronal space : 236 s

Path finding : 850 s.

To eventually speed up the path finding process, np.roll() (used extensively in finding functions) must be written with Numba support. This, after discussion, will possibly be set as the prior target for next stage of work.

We'll also need to implement an animating function to generate small videos visualizing the process or suite of pictures. And preparation for robot system should also start.

PS : Values in this report may not be very exact as the condition (temperature) of my laptop affects the computational power.