

Introduction

Shizeng

1. Project Overview

Through this project, I learned the basic knowledge of motion planning systematically. The project includes seven parts and each part includes theoretical study and practical operations. This project includes three main parts: Front-end Path finding, Back-end trajectory generation and MDP&MPC applications.

For the operating system, it is mainly based on the Matlab and ROS/C++. Below, I will introduce three parts separately.

2. Front-end Path finding

This part includes search-based path finding and sampling-based path finding.

Firstly, for the search-based path finding, Dijkstra and A* algorithms have been introduced.

Actually these two algorithms are similar and compared with Dijkstra Algorithm, A* algorithm has a heuristic function so that it may get the goal more quickly. I use Matlab and ROS/C++ to implement A* algorithm. Below are the result figures:

The dark green points are the obstacles and the light points are the finding path.

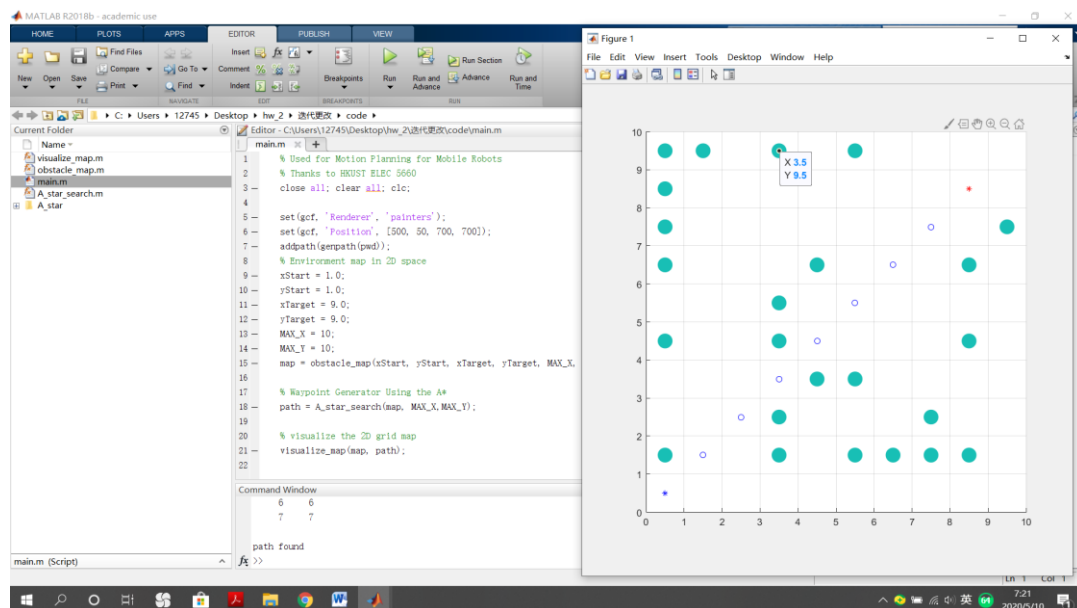


Fig1: A* algorithm Matlab result

The implement on ROS is three dimensional and it is based on a random map which is created by the Rviz and the concrete code implementation can be found in my Github.

Secondly, for the sampling-based path finding, rapidly-exploring random tree(RRT) has been introduced. The Matlab implement is shown below, the black part is obstacles and the green branches are the finding points. The blue line is the final path.

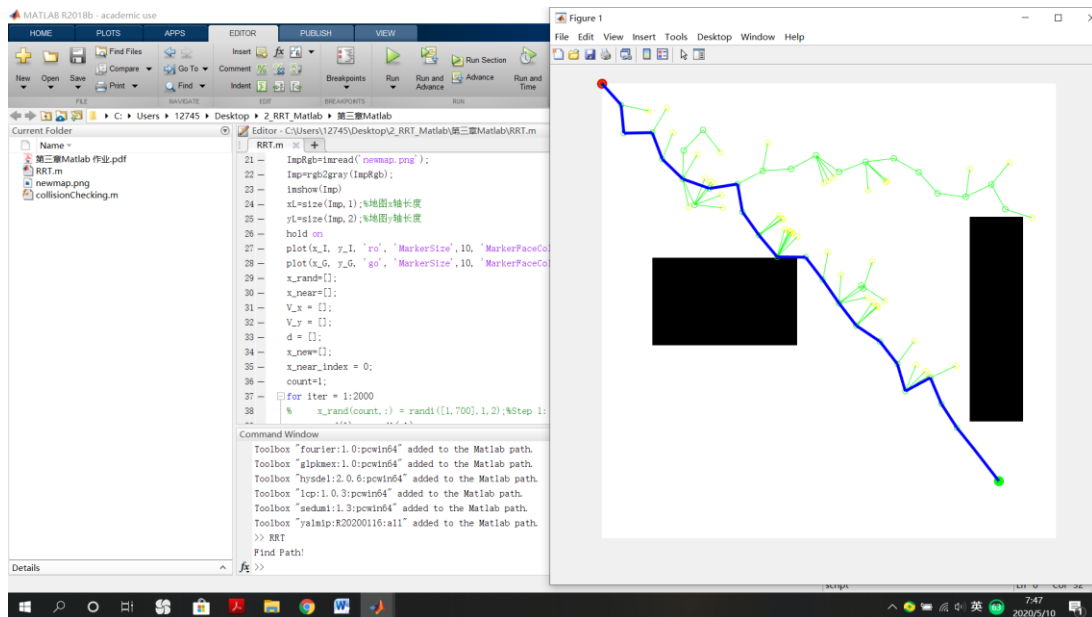


Fig3: RRT algorithm Matlab result

3. End-front Path generation

This part includes minimum snap trajectory generation and soft & hard constrained trajectory optimization.

Firstly, for the minimum snap trajectory generation, it is aim to smooth the trajectory then the mobile robot can have a better motion and save energy.

The basic idea is to use the polynomial segments to express whole trajectory. Because of the minimum snap, so here we choose 7th order. After that we add constraints to the key points like start and end points for their positions, velocities and accelerations. Finally, we apply convex optimization to find the optimal trajectory. Below is the implement on Matlab.

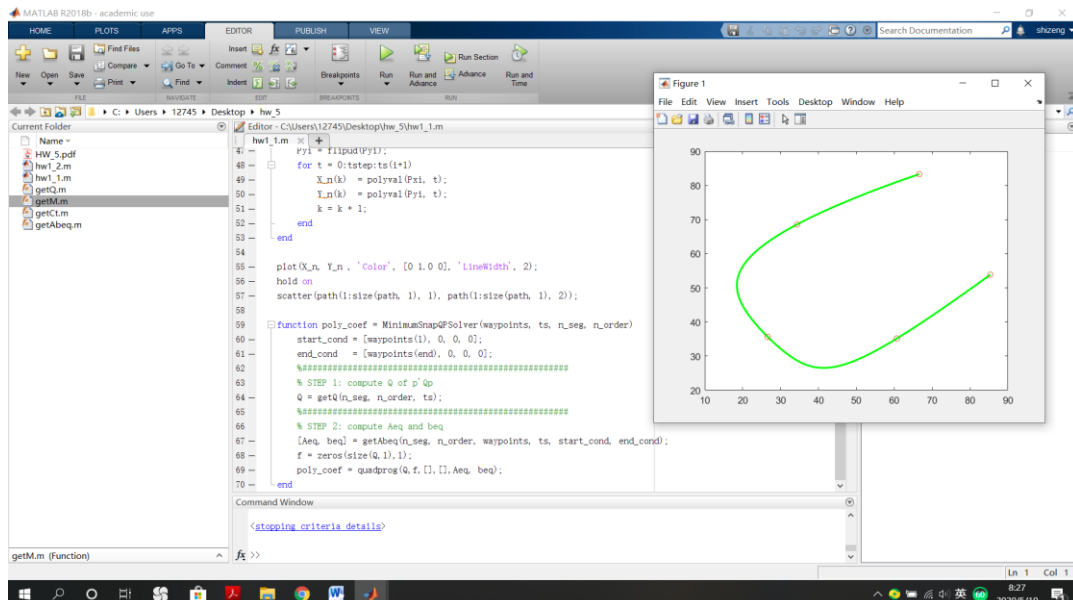
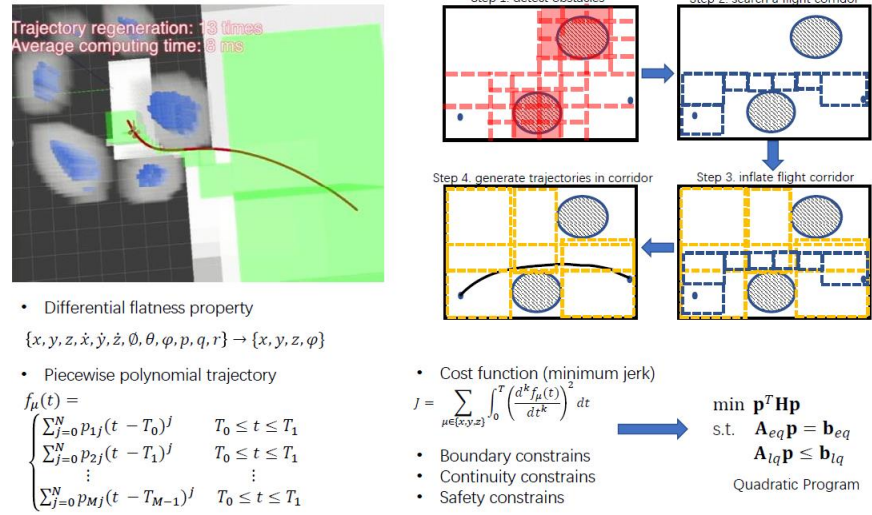


Fig4: minimum snap trajectory generation

Secondly, for the minimum snap trajectory generation, it only constrain intermediate

waypoints of the trajectory should pass. It has no constraints on the trajectory so that the trajectory may “overshoot”.

The basic idea to do this is to apply corridor-based smooth trajectory generation and here we also use Bernstein polynomial basis.



Online generation of collision-free trajectories for quadrotor flight in unknown cluttered environments, Jing Chen et al.

Fig5: corridor-based smooth trajectory generation

- Use **Bernstein** polynomial basis.
 - Change to basis of the trajectory from **monomial** polynomial to **Bernstein** polynomial
- $$P_j(t) = p_j^0 + p_j^1 t + p_j^2 t^2 + \dots + p_j^n t^n \rightarrow B_j(t) = c_j^0 b_n^0(t) + c_j^1 b_n^1(t) + \dots + c_j^n b_n^n(t) = \sum_{i=0}^n c_j^i b_n^i(t)$$

$$b_n^i(t) = \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i}$$
- Bézier curve
- Bézier curve is just a special polynomial, it can be mapped to monomial polynomial by:
 $\mathbf{p} = \mathbf{M} \cdot \mathbf{c}$. And all previous derivations still hold.

For 6 order:
$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -6 & 6 & 0 & 0 & 0 & 0 & 0 \\ 15 & -30 & 15 & 0 & 0 & 0 & 0 \\ -20 & 60 & -60 & 20 & 0 & 0 & 0 \\ 15 & -60 & 90 & -60 & 15 & 0 & 0 \\ -6 & 30 & -60 & 60 & -30 & 6 & 0 \\ 1 & -6 & 15 & -20 & 15 & -6 & 1 \end{bmatrix}$$

Fig6: Bernstein polynomial basis

The implement is on the Matlab and the basic idea is to transform the polynomial segments to the Bernstein polynomial basis. Then add constraints and use optimal solvers.

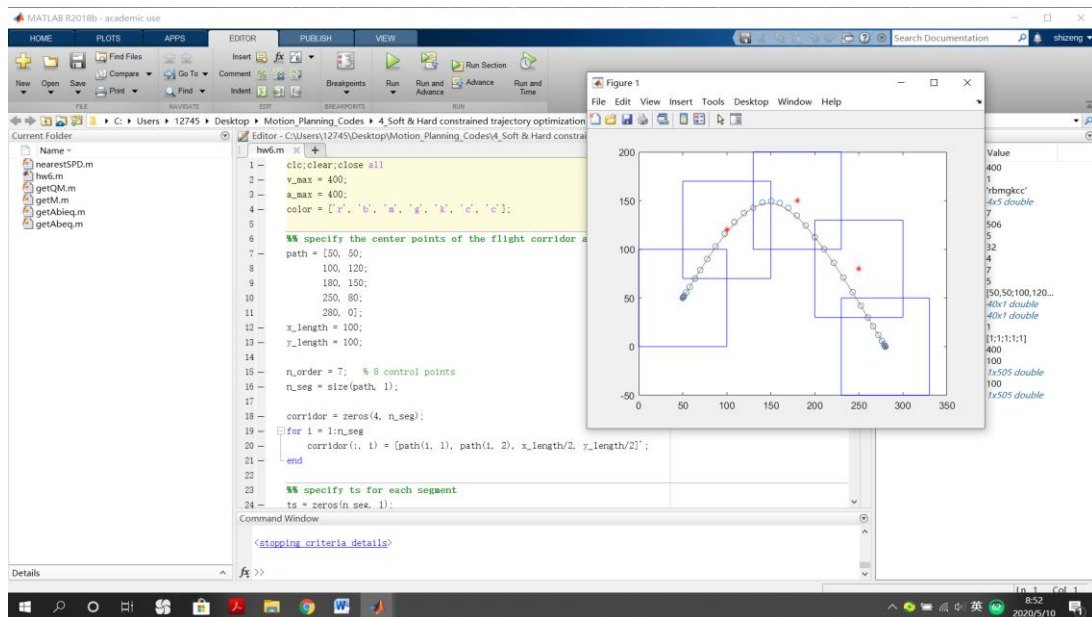


Fig7: Soft & Hard constrained trajectory optimization

4. MDP&MPC applications

After Front-end Path finding and Back-end trajectory generation, I try to simply apply Markov Decision Process-Based Planning and Model Predictive Control for Robotics Planning.

Firstly, for the Markov Decision Process-Based Planning, I use Python to implement a demo of a race which tries to find an optimal trajectory to get the final goal.

Here I apply Real Time Dynamic Programming, and the algorithm flow is shown below:

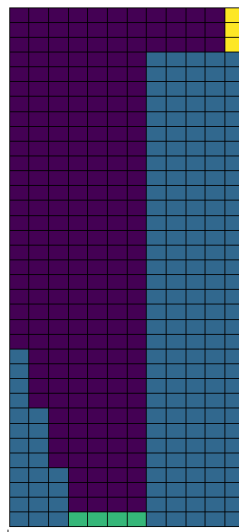
RTDP algorithm:

- ① Initialize G values of all states to admissible values;
- ② Follow greedy policy picking outcomes at random until goal is reached;
- ③ Backup all states visited on the way;
- ④ Reset to x_s and repeat 2-4 until all states on the current greedy policy have Bellman errors $< \Delta$, where $\Delta(x_k) = \|G(x_k) - G(x_{k+1})\|$;

Advantages:

- ① very efficient alternative to Value Iteration
- ② does NOT compute values of all states
- ③ focuses computations on states that are relevant

For this demo , X is the part that allow path, X_i is the start point, X_f is the final point, U is the acceleration costs, L is the moving costs and θ is the possibility whether to go or not. 0.9 to go and 0.1 to stop.



Grid Map

- ① $X = \{(x, y) \mid 0 \leq x \leq 11, 0 \leq y \leq 34\}$
- ② $X_I = \{\text{green grids}\}$
 $X_F = \{\text{yellow grids}\}$
- ③ $U = \{(\ddot{x}, \ddot{y}) \mid \ddot{x} \in \{0, \pm 1\}, \ddot{y} \in \{0, \pm 1\}\}$
- ④ $\Theta = \{\theta_1, \theta_2\}$
 - $\theta_1: f(\mathbf{x}_{k+1}, \mathbf{x}_k, \mathbf{u}_k) = \mathbf{x}_k \quad p_1 = 0.1$
 - $\theta_2: f(\mathbf{x}_{k+1}, \mathbf{x}_k, \mathbf{u}_k) = \mathbf{x}_{k+1} \quad p_1 = 0.9$
- ⑤ $l(\mathbf{x}_k, \mathbf{x}_k, \theta_k) = -1$
- ⑥ Find an optimal plan from X_I to X_F

Fig8: demo conditions

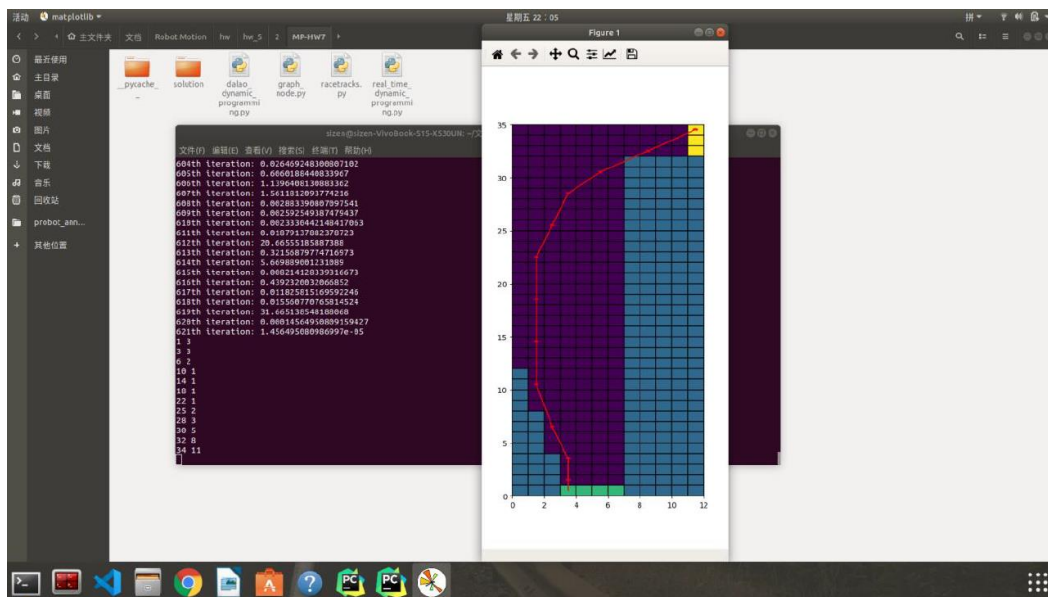
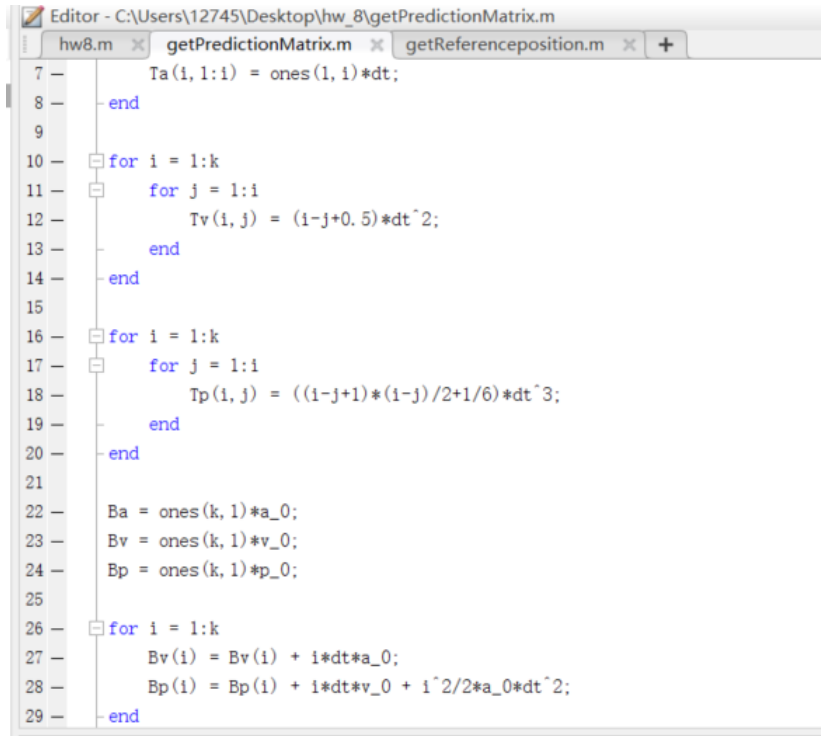


Fig9: demo result

Secondly, for the Model Predictive Control for Robotics Planning, I try to keep the trajectory follow the spirals that change in real time.

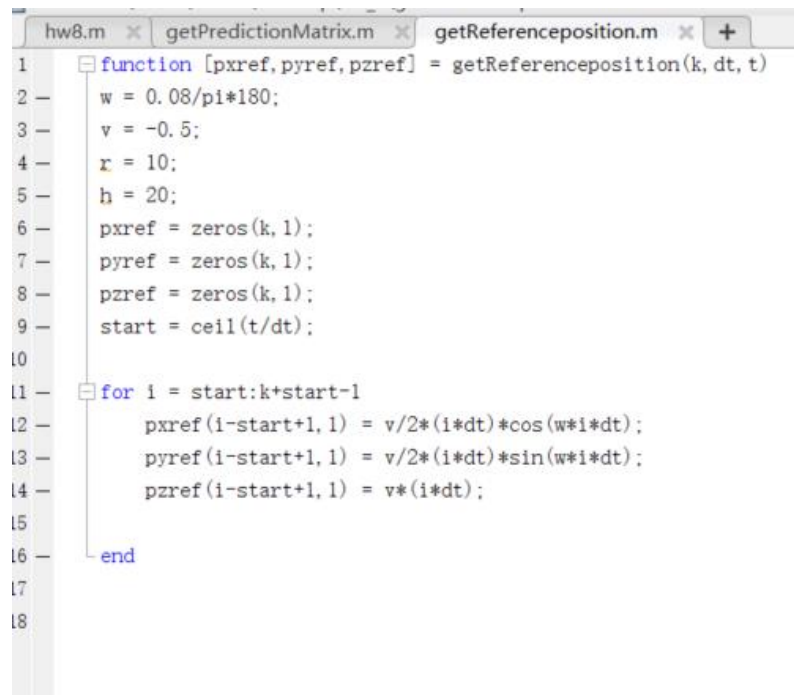
The basic idea is to build prediction matrix and get reference position. The function of `getPredictionmatrix` is to build the predict integrator and the function of `getReferenceposition` is to get the position of spirals.



```

Editor - C:\Users\12745\Desktop\hw_8\getPredictionMatrix.m
hw8.m  getPredictionMatrix.m  getReferenceposition.m  +
7  Ta(1,1:i) = ones(1,i)*dt;
8  end
9
10 for i = 1:k
11     for j = 1:i
12         Tv(i,j) = (i-j+0.5)*dt^2;
13     end
14 end
15
16 for i = 1:k
17     for j = 1:i
18         Tp(i,j) = ((i-j+1)*(i-j)/2+1/6)*dt^3;
19     end
20 end
21
22 Ba = ones(k,1)*a_0;
23 Bv = ones(k,1)*v_0;
24 Bp = ones(k,1)*p_0;
25
26 for i = 1:k
27     Bv(i) = Bv(i) + i*dt*a_0;
28     Bp(i) = Bp(i) + i*dt*v_0 + i^2/2*a_0*dt^2;
29 end
  
```

Fig10: `getPredictionmatrix`



```

hw8.m  getPredictionMatrix.m  getReferenceposition.m  +
1  function [pxref,pyref,pzref] = getReferenceposition(k,dt,t)
2  w = 0.08/pi*180;
3  v = -0.5;
4  r = 10;
5  h = 20;
6  pxref = zeros(k,1);
7  pyref = zeros(k,1);
8  pzref = zeros(k,1);
9  start = ceil(t/dt);
10
11 for i = start:k*start-1
12     pxref(i-start+1,1) = v/2*(i*dt)*cos(w*i*dt);
13     pyref(i-start+1,1) = v/2*(i*dt)*sin(w*i*dt);
14     pzref(i-start+1,1) = v*(i*dt);
15 end
16
17
18
  
```

Fig11: `getReferenceposition`

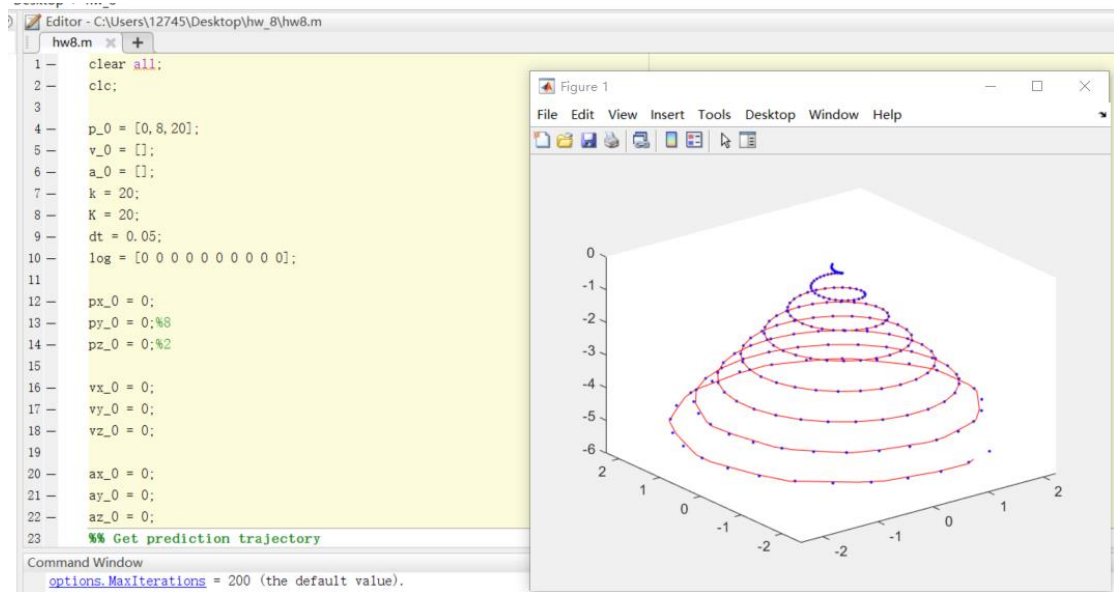


Fig12: result