Dynamic Programming

Manoj Kumar Reddy Manchala
Department of Mechanical and Aerospace Engineering
University of California San Diego
La Jolla, U.S.A.
mmanchala@ucsd.edu

Nikolay Atanasov
Department of Electrical and Computer Engineering
University of California San Diego
La Jolla, U.S.A.
natanasov@eng.ucsd.edu

***Abstract*—This project addresses the challenge of trajectory tracking in autonomous robotic systems, a crucial aspect of robotics that demands high precision and adaptability in dynamic environments. Utilizing the Generalized Policy Iteration (GPI) algorithm, we developed a robust control policy designed to guide a differential-drive robot to follow a specified trajectory while avoiding obstacles. The GPI framework was chosen for its ability to iteratively improve control policies based on a systematic evaluation and improvement process, leveraging a discretized state and control space to manage the continuous nature of the robotic control problem.**

The implementation involved defining a value function within a grid-based framework and calculating transition probabilities and stage costs to estimate the effectiveness of potential actions. Through multiple iterations of policy evaluation and improvement, the algorithm refined its strategy to minimize a composite cost function that balances trajectory deviation and control effort. The project utilized Python, leveraging libraries such as NumPy for numerical operations and Ray for parallel computing, facilitating efficient computation of the GPI algorithm.

Results demonstrate that the GPI algorithm effectively reduces trajectory tracking errors and avoids collisions with static obstacles, providing a reliable and efficient method for real-time robotic control. This approach not only enhances the autonomous navigation capabilities of robots but also offers insights into the application of advanced control theories in practical robotics systems.

***Index Terms*—Certainty Equivalent Control, Grid discretization, GPI, CEC**

I. Introduction

Trajectory tracking is a pivotal challenge in robotics, essential for ensuring precise adherence to a predetermined path in diverse applications ranging from autonomous vehicles navigating urban environments to robots operating in industrial settings. The complexity of the problem arises from the need to maintain accuracy despite dynamic obstacles, unpredictable terrains, and inherent system uncertainties.

In addressing this challenge, this project evaluates two distinct computational strategies: Certainty Equivalent Control (CEC) and Generalized Policy Iteration (GPI). CEC is a simplified control approach where the system dynamics are presumed to be known and certain, thus reducing a stochastic control problem to a deterministic one by assuming no uncertainty in model parameters or external disturbances. Although CEC simplifies problem-solving by ignoring uncertainties, it might not adequately handle real-world unpredictabilities.

On the other hand, GPI, derived from dynamic programming principles, offers a more robust solution by iteratively improving the policy based on a value function that evaluates the long-term costs of actions. GPI involves discretizing the state and control spaces and calculating transition probabilities and stage costs, allowing for the development of a control policy that optimally reduces trajectory deviations and manages control efforts efficiently.

The project utilizes a differential-drive robot as the test platform, focusing on the algorithmic implementation in Python with extensive use of NumPy for matrix operations and Ray for parallel computing, enhancing the computational efficiency of our GPI implementation. Through this exploration, the project aims to demonstrate the comparative effectiveness of CEC and GPI in managing the complexities of real-time trajectory tracking in robotic systems.

II. Problem Statement

The fundamental challenge addressed in this project is the trajectory tracking problem for a differential-drive robot. Mathematically, the problem can be framed within the context of stochastic control, where the robot must follow a predefined trajectory $(r_t)_{t\geq 0}$, defined in the two-dimensional space, with high precision and reliability despite the presence of uncertainties and external disturbances.

A. System Dynamics

The robot's state at time t, denoted as x_t, comprises its position $p_t \in \mathbb{R}^2$ and orientation $\theta_t \in [-\pi, \pi)$. The control inputs, u_t, include linear velocity $v_t \in \mathbb{R}$ and angular velocity $\omega_t \in \mathbb{R}$, affecting the robot's state according to the following discrete-time stochastic model:

$$\begin{aligned} x_{t+1} &= f(x_t, u_t, w_t) \\ &= \begin{bmatrix} p_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} p_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} \Delta\cos(\theta_t) & 0 \\ \Delta\sin(\theta_t) & 0 \\ 0 & \Delta \end{bmatrix} \begin{bmatrix} v_t \\ \omega_t \end{bmatrix} + w_t, \end{aligned}$$

where $w_t \sim \mathcal{N}(0, \mathrm{diag}(\sigma^2))$ represents the motion noise with standard deviation σ, and Δ is the discretization time step.

B. Objective Function

The objective of the trajectory tracking problem is to minimize the cumulative cost over an infinite horizon, defined by the deviation from the reference trajectory and the control

effort. The cost function at each time step is a quadratic function of the state error and control input:

$$J(u) = \sum_{t=0}^{\infty} \gamma^t \left(\|p_t - r_t\|_Q^2 + q(1 - \cos(\theta_t - \alpha_t)) + \|u_t\|_R^2 \right), \tag{1}$$

where $\gamma \in (0, 1)$ is the discount factor, Q and R are positive definite matrices representing the weights on the positional error and control effort, respectively, and $q > 0$ is the weight on the orientation error.

C. Constraints

The robot is required to navigate within a bounded environment while avoiding collisions with obstacles. The admissible control set, U, is defined by the physical capabilities of the robot:

$$U = \{(v_t, \omega_t) \mid 0 \leq v_t \leq 1, -1 \leq \omega_t \leq 1\}. \tag{2}$$

Additionally, the robot must avoid defined obstacle regions within the environment, ensuring safe navigation without collisions.

This problem is addressed through two main strategies in this project: Certainty Equivalent Control (CEC) and Generalized Policy Iteration (GPI), each of which provides a method to derive a suitable control policy under different assumptions and computational paradigms.

III. Technical Approach

A. *Receding-Horizon Certainty Equivalent Control (CEC)*

Certainty Equivalent Control (CEC) is a control strategy that simplifies the control problem by assuming deterministic system dynamics, thus treating the problem as if there are no uncertainties or disturbances. This approach allows for the formulation of an optimization problem that can be solved using nonlinear programming (NLP) techniques. In this project, we apply a receding-horizon approach, also known as Model Predictive Control (MPC), to solve the trajectory tracking problem for a differential-drive robot.

1) Nonlinear Programming Formulation: The receding-horizon CEC approach involves solving an optimization problem over a finite prediction horizon T at each time step. The optimization problem is formulated to minimize a cost function that accounts for both the tracking error and the control effort. The steps are as follows:

a) Optimization Variables: The optimization variables include the control inputs over the prediction horizon:

$$\mathbf{u} = \begin{bmatrix} u_0^T & u_1^T & \cdots & u_{T-1}^T \end{bmatrix}^T \tag{3}$$

where $u_k = \begin{bmatrix} v_k & \omega_k \end{bmatrix}^T$ represents the linear and angular velocities at time step k.

b) Objective Function: The objective function is designed to minimize the cumulative cost over the prediction horizon:

$$J = \sum_{k=0}^{T-1} \gamma^k \left(e_k^T Q e_k + q(1 - \cos(\theta_k - \alpha_k))^2 + u_k^T R u_k \right) \tag{4}$$

where γ is the discount factor, Q and R are positive definite matrices weighting the state and control costs, and q is a scalar penalty on the orientation error. The state error e_k is defined as the difference between the current state and the reference trajectory.

c) Constraints: The optimization is subject to several constraints:

- **System Dynamics:**

$$x_{k+1} = f(x_k, u_k) = x_k + \begin{bmatrix} \Delta t \cos(\theta_k) & 0 \\ \Delta t \sin(\theta_k) & 0 \\ 0 & \Delta t \end{bmatrix} u_k \tag{5}$$

- **Collision Avoidance:**

$$\sqrt{(x_{k+1} - x_{\text{obs}})^2 + (y_{k+1} - y_{\text{obs}})^2} > r_{\text{obs}} \tag{6}$$

  where $(x_{\text{obs}}, y_{\text{obs}}, r_{\text{obs}})$ represents the position and radius of the obstacles.
- **Control Input Bounds:**

$$v_{\min} \leq v_k \leq v_{\max}, \quad \omega_{\min} \leq \omega_k \leq \omega_{\max} \tag{7}$$

2) Implementation Details: The CEC method is implemented in Python using the CasADi library, which provides tools for automatic differentiation and optimization. Below is the implementation code presented as pseudocode:

The implementation defines the optimization variables, constraints, and objective function in the context of the receding-horizon CEC framework. The CasADi solver is used to solve the optimization problem at each time step, ensuring that the control inputs minimize the cost function while satisfying the constraints.

3) Theoretical and Analytical Details: The CEC approach provides a tractable way to handle the trajectory tracking problem by treating the control inputs as deterministic and using a receding horizon to continuously update the control strategy based on the current state. The NLP formulation ensures that the control actions are optimized over a finite horizon, which is then shifted forward at each time step, providing a balance between computational efficiency and control performance.

B. *Generalized Policy Iteration (GPI)*

Generalized Policy Iteration (GPI) is a robust method for solving Markov Decision Processes (MDPs), which involves iterative processes of policy evaluation and policy improvement. In this project, GPI is utilized to optimize the control policy over a discretized state and control space for a differential-drive robot.

Algorithm 1 Certainty Equivalent Control (CEC)

Require: Current state x_t, Reference state r_t

Initialize control inputs $u \leftarrow \text{zeros}(2T)$
Initialize state errors $e \leftarrow \text{zeros}(3(T+1))$
Set $X_k \leftarrow x_t$
Compute initial error $e_0 \leftarrow x_t - r_t$
Initialize objective function $J \leftarrow \text{cost\_function}(e_0, u[0:2], 0)$
Initialize constraints with initial error $e[0:3] = e_0$
for $k = 0$ to $T-1$ **do**
  Extract control inputs $u_k \leftarrow u[2k:2k+2]$
  Compute next state $X_{k+1} \leftarrow \text{car\_next\_state\_casadi}(X_k, u_k)$
  Normalize orientation $X_{k+1}[2] \leftarrow \text{normalize\_angle}(X_{k+1}[2])$
  Compute reference state $r_{k+1} \leftarrow \text{lissajous}(t+k+1)$
  Compute next error $e_{k+1} \leftarrow X_{k+1} - r_{k+1}$
  Normalize orientation error $e_{k+1}[2] \leftarrow \text{normalize\_angle}(e_{k+1}[2])$
  Update objective function $J \leftarrow J + \text{cost\_function}(e_{k+1}, u_k, k)$
  Append error constraint $e[3(k+1):3(k+2)] = e_{k+1}$
  Append collision avoidance constraints
  $X_k \leftarrow X_{k+1}$
end for
Define NLP problem with objective J and constraints
Solve NLP using CasADi's solver
Extract optimal control input $u^* \leftarrow u[0:2]$
return u^* =0

1) State Discretization: The state space, comprising position and orientation errors, is discretized into a grid structure to handle the continuous nature of the state variables. This discretization facilitates the use of a tabular representation for the value function, allowing the application of dynamic programming techniques.

a) Discretized State Space:

$$\begin{aligned} \text{ex\_space} &= \text{linspace}(-3, 3, 21), \\ \text{ey\_space} &= \text{linspace}(-3, 3, 21) \\ \text{eth\_space} &= \text{linspace}(-\pi, \pi, 40) \end{aligned}$$

2) Collision Avoidance: Collision avoidance is integrated into the GPI by penalizing transitions that lead to states within a predefined proximity to obstacles. This is achieved by augmenting the stage costs with high penalties for near-collision states.

a) Stage Cost Function:

$$\text{stage\_cost} = \mathbf{e}^T Q \mathbf{e} + q(1 - \cos(\theta))^2 + \mathbf{u}^T R \mathbf{u} \tag{8}$$

where $\mathbf{e}$ is the state error, θ is the orientation error, $\mathbf{u}$ is the control input, and Q, q, and R are weighting matrices/scalar.

3) Value Function: The value function, $V(s)$, quantifies the expected return (or cost) starting from state s, following a certain policy π. For GPI, we iteratively update this value function to reflect more accurate estimations of returns as policies are improved.

4) Scalability Management: To manage the large state space resulting from discretization, the GPI algorithm leverages parallel computing techniques provided by the Ray library. This allows the decomposition of the policy evaluation and improvement steps across multiple processors, significantly speeding up the computations.

a) Precomputation of Transition Matrices and Stage Costs: Precomputed transition matrices and stage costs are used to expedite the GPI computations. These matrices are calculated in advance and stored, reducing the runtime overhead during policy iteration.

Algorithm 2 Precompute Transition Matrices and Stage Costs

Require: Discretized state and control spaces, reference trajectory

for each time step t **do**
  Compute reference states ref_cur and ref_next
  for each state in the discretized state space **do**
    for each control input in the discretized control space **do**
      Compute next state mean
      Compute neighbors and probabilities
      Store in transition matrix
      Compute stage cost
      Store in stage cost matrix
    end for
  end for
end for
Save transition matrices and stage costs =0

5) Theoretical and Analytical Details: The GPI algorithm consists of two main steps: policy evaluation and policy improvement. GPI optimizes the control policy through iterative policy evaluation and improvement. The policy evaluation step computes the value function for a given policy, while the policy improvement step updates the policy based on the computed value function. By discretizing the state and control spaces and leveraging parallel computing, the algorithm efficiently handles the large state space, ensuring robust and optimal control performance for trajectory tracking in dynamic environments.

a) Policy Evaluation: In the policy evaluation step, the value function $V(s)$ for a given policy π is computed. The value function represents the expected cumulative cost when starting from state s and following policy π.

$$V(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k c(s_k, \pi(s_k)) \mid s_0 = s\right] \tag{9}$$

In practice, this infinite sum is approximated over a finite horizon T.

b) Policy Improvement: In the policy improvement step, the policy π is updated to minimize the expected cumulative cost based on the current value function $V(s)$.

$$\pi'(s) = \arg\min_{u \in \mathcal{U}} \left[c(s,u) + \gamma \sum_{s'} P(s' \mid s,u) V(s') \right] \quad (10)$$

6) Implementation Details: The GPI method is implemented in Python, utilizing NumPy for numerical operations and Ray for parallel computing. Below is the pseudocode for the GPI algorithm, including policy evaluation and improvement steps:

Algorithm 3 Generalized Policy Iteration (GPI)

Require: Value function V, Transition matrices, Stage costs
Initialize policy π
for each iteration **do**
  Policy Evaluation:
  for each time step t **do**
    for each state $(ex, ey, etheta)$ **do**
      Initialize minimum cost to infinity
      for each control input (v, w) **do**
        Compute total cost using stage cost and transition probabilities
        if total cost ¡ minimum cost **then**
          Update minimum cost
          Update policy with best control input
        end if
      end for
      Update value function with minimum cost
    end for
  end for
  Policy Improvement:
  for each time step t **do**
    for each state $(ex, ey, etheta)$ **do**
      Compute best action from value function
      Update policy
    end for
  end for
end for
return Optimal policy π =0

IV. Results

A. CEC

*1) **CEC - Before Tuning**:* The following image represents the trajectory tracking before tuning the parameters.

*2) **CEC - Higher Q**:* Higher value of Q forces the position error to be less and we can see from the plot that the trajectory is followed well. But it is now intersecting obstacle, which means the ability to change the control is dominated by the necessity to track the reference trajectory.

*3) **CEC - Higher x component, lower y component**:* In this case, the component of Q corresponding to x position is kept high, but coomponent corresponding to y position is kept low so that the tracking in y direction will not be forced as much, which can be seen from the image.

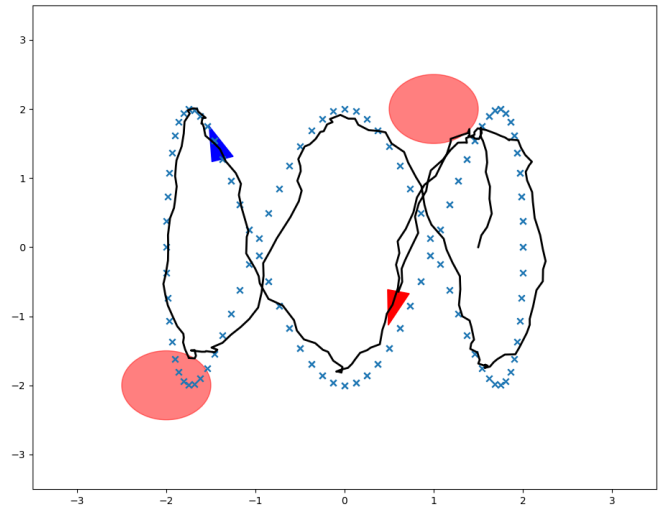

Fig. 1. Agent's path through the environment

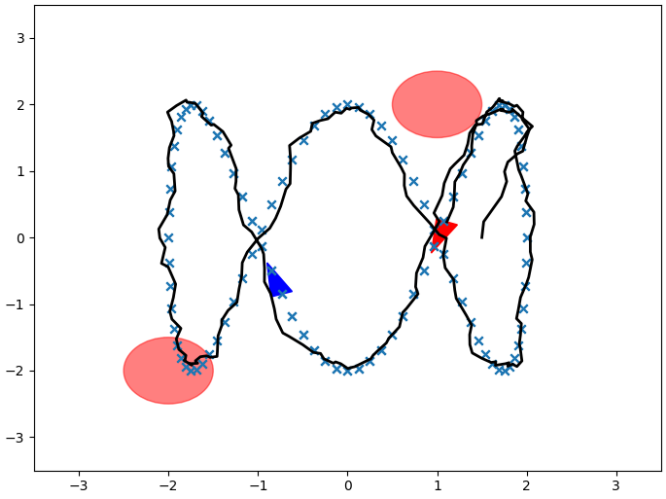

Fig. 2. Agent's path through the environment

*4) **CEC - Higher x component in R**:* In this case, the component of R corresponding to velocity control is kept high, but coomponent corresponding to angular velocity control is kept low. We can see that it now avoids obstacles, but it also does not track the trajectory near the turns.

*5) **CEC - Higher y component in Q**:* In this case te y component of Q is increased compared to the previous case, so as to reduce the error in tracking along the y direction near the turns. We can see that the error is indeed reduced.

*6) **CEC - Lower q**:* In this case the value of q, which corresponds to the error in the heading direction is reduced,

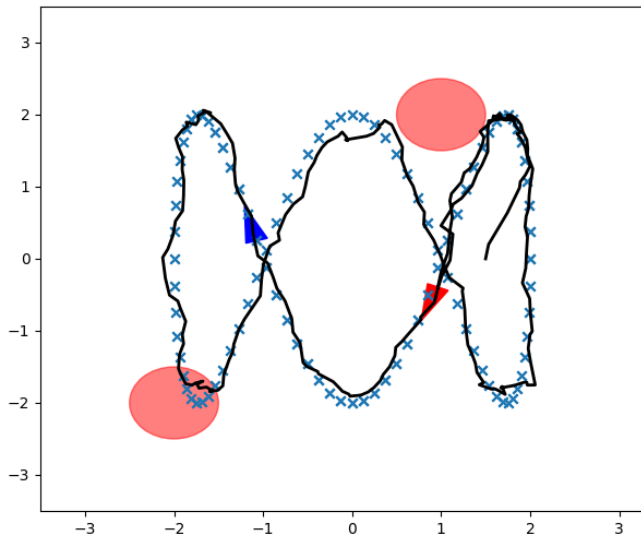

Fig. 3. Agent's path through the environment

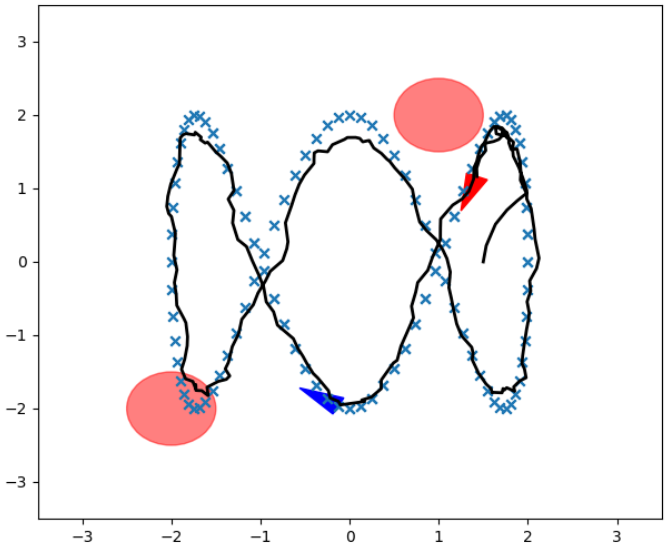

Fig. 5. Agent's path through the environment

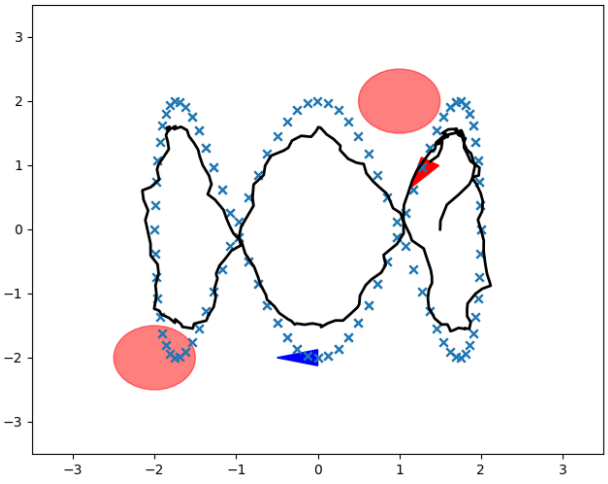

Fig. 4. Agent's path through the environment

which means there is now some room for error in the heading direction, which is necessary as near the obstacles, the heading direction error will be high. We can see that this improved the trajectory.

7) ***CEC - Further Tuning****:* In this case the value of q, which corresponds to the error in the heading direction is reduced, which means there is now some room for error in the heading direction, which is necessary as near the obstacles, the heading direction error will be high. We can see that this improved the trajectory.

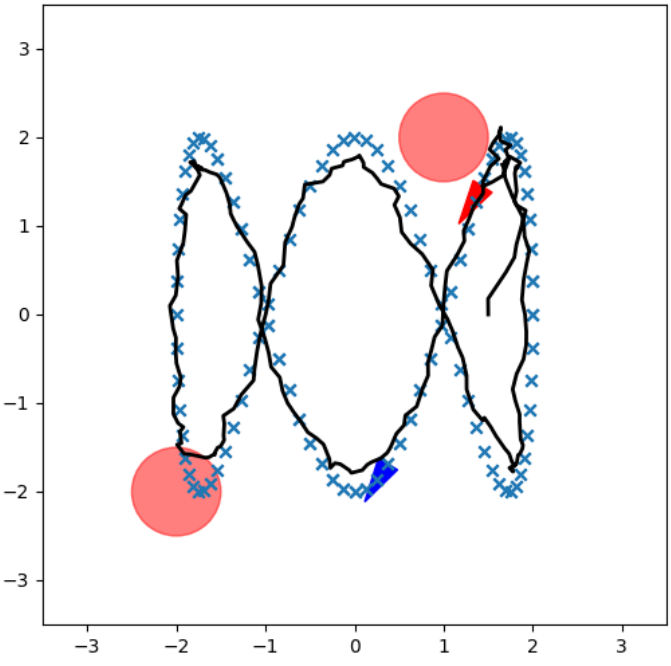

Fig. 6. Agent's path through the environment

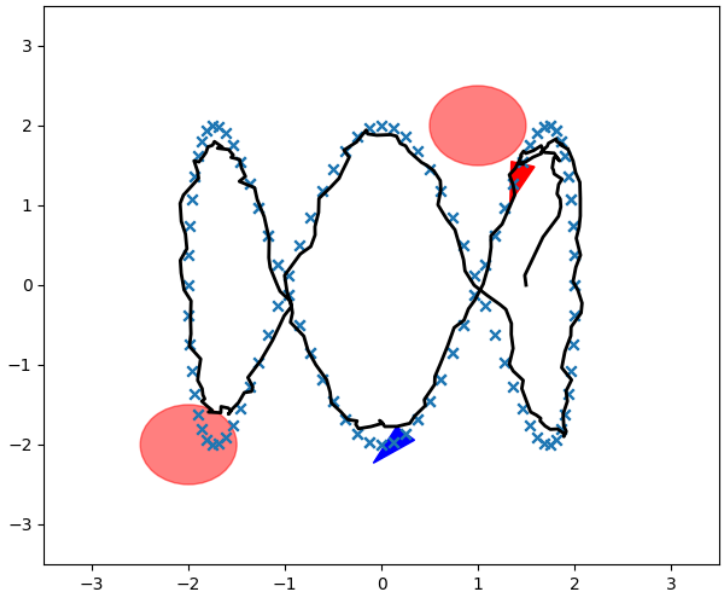

Fig. 7. Agent's path through the environment

B. *GPI*

The GPI algorithm took too long to run and the results were not good. Due to time constraints could not tune the algorithm even though the transition matrix and stage cost are pre computed and stored on the disk, the policy evaluation and policy improvement steps were also consuming significant memory and also time. I have vectorized the algorithm and reduced the computation times for transition matrix and stage cost to less than 2 minutes. But the same could not be achieved for policy evaluation and improvement steps.

C. *Discussion*

When compared to GPI, the CEC algorithm is computaionally very less expensive.In the GPI algorithm due to discretization, huge matrices need to be computed which is both computationally intensive and also memory intensive. Where as CEC is a constrained optimization problem. The computation time for CEC is around 10-15 seconds depending on the horizon, but in the case of GPI, the computation of transition matrices and stage costs itself takes 2-3 minutes with parallalization.

Although the key difference is that the trajectory of the GPI algorithm is smooth compared to CEC due to the discretization. Also CEC has its limitations of needing to tune multiple parameters. Whereas GPI can have identity initialization and will just need more iterations for the policy to converge.