ENPM667

Control of Robotic Systems

Term project on:

Grey Wolf Optimizer based tuning of Hybrid LQR-PID controller for foot trajectory control

Under the Guidance of:

Dr. Waseem Malik

University of Maryland, College Park

Submitted by
Raghav Agarwal
115078055

Toyas Dhake
116507271

Abstract

Quadruped robots have a complex construction which results in increased struggle when designing a stable controller for the robot. Legged locomotion results from complex, high-dimensional, nonlinear, dynamically coupled interactions between an organism and its environment. In this report we design and optimize an effective hybrid control by combining PID and LQR controllers. In this report the hybrid LQR-PID controller is tuned using a Grey Wolf optimizer which is then compared with the results obtained from Particle Swarm Optimization. The principal goal of this report is to compute the LQR controller parameters (Q and R weight matrices) and the PID controller gains(kp, ki and kd). Initially, the dynamical equations are obtained by turning the Euler-Lagrange crank and the system is then linearized to obtain the state space model of the manipulator. Later, the hybrid PID-LQR control system is designed and optimal values which guarantee best trajectory tracking are obtained.

Table of Content

List of figures

Introduction

Today mobile manipulation robots are a major research area in the field of robotics. These types of robots are being implemented in a multitude of environments like household, health care, military operations, for study and research purposes in extreme conditions like around a nuclear disaster site. So, the mobile manipulator can be used for a large variety of purposes. A quadruped robot can be used to access places that are not accessible by humans or by wheeled mobile robots. Taking inspiration from biology where several mechanisms exist to perform locomotion, many legged robots are built. They are different, from their work goal to design. Bringing ideas from existing mechanisms in nature, we can develop a machine that can be used to help in search and rescue during a natural calamity. The complexity of the system makes it difficult to control and requires the development of various controllers. We use PID controller as it is one of the extensively used controllers in many engineering applications because of its effectiveness and ease of implementation. The job of a PID controller is to force feedback to match a setpoint. Sometimes error between feedback and setpoint is caused by a setpoint change, but in most applications the setpoint is not adjusted much. More often, error in a loop is caused by disturbances in measured feedback. In designing the PID controller, the gains were determined by trial and error methods. We also implement a LQR controller. The Linear Quadratic Regulator (LQR) is a well-known method that provides optimally controlled feedback gains to enable the closed-loop stable and high performance design of systems. Similarly, in the design of the LQR controller, we select the state (Q) and control (R) weighting matrices. Finally, in order to adjust the PID gains we implement Grey Wolf Optimizer and Particle Swarm Optimization techniques. The flowchart of this report including modelling the system, linearizing the system, designing and optimising the controllers is given below.

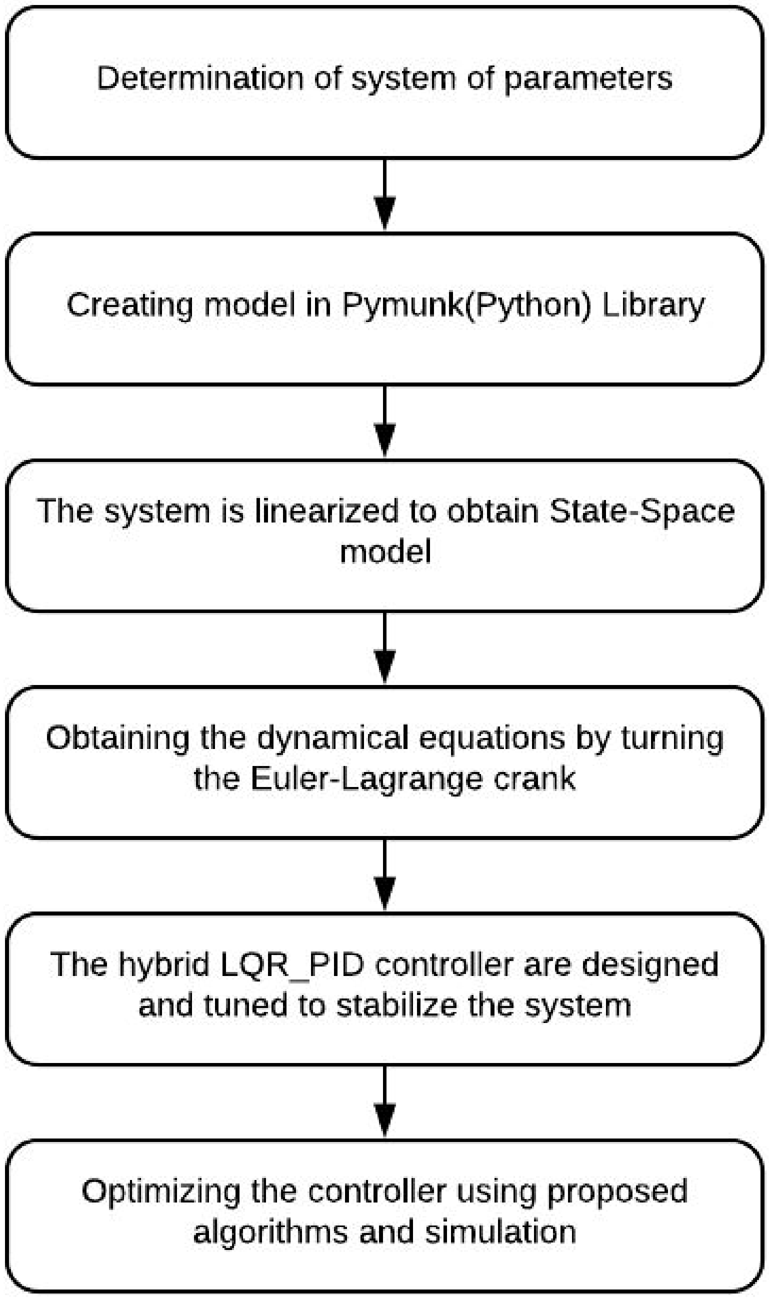

Determination of system of parameters
Creating model in Pymunk(Python) Library
The system is linearized to obtain State-Space model
Obtaining the dynamical equations by turning the Euler-Lagrange crank
The hybrid LQR_PID controller are designed and tuned to stabilize the system
Optimizing the controller using proposed algorithms and simulation

Figure 1: Flowchart of the report

Obtaining the equation of motion of the system

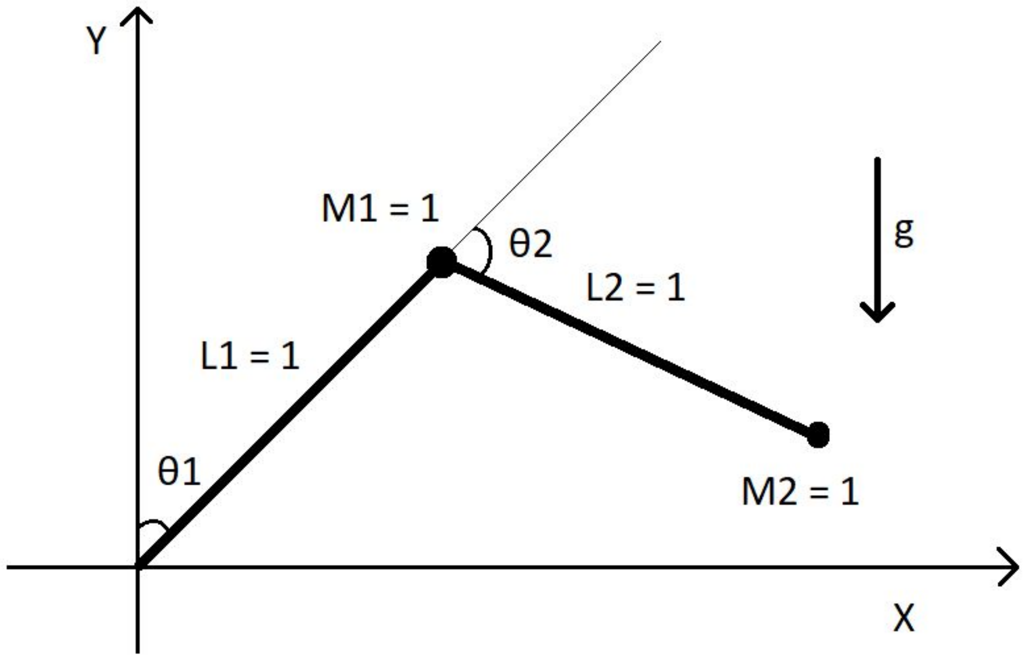


Figure 2: Orientation of the leg

A. Dynamical equations for 2 DOF manipulator

From the orientation given in figure 2 we get the following equations -

$$x_1 = L_1 \sin\theta_1$$
$$y_1 = L_2 \cos\theta_1$$
$$x_2 = L_1 \sin\theta_1 + L_2 \sin(\theta_1 + \theta_2)$$
$$y_2 = L_1 \cos\theta_1 + L_2 \cos(\theta_1 + \theta_2)$$

So, Kinetic energy can be computed as,

$$Kinetic\ Energy\ =\ \tfrac{1}{2}M_1\dot{x}_1{}^2\ +\ \tfrac{1}{2}M_1\dot{y}_1{}^2 +\ \tfrac{1}{2}M_2\dot{x}_2{}^2\ +\ \tfrac{1}{2}M_2\dot{y}_2{}^2$$

Here we have $cos(\theta_1)\ as\ c_1,\ cos(\theta_2)\ as\ c_2\ , sin(\theta_1)\ as\ s_1\ and\ sin(\theta_2)\ as\ s_2.$

$$= \tfrac{1}{2}M_1[L_1{}^2cos^2\theta_1\dot{\theta}_1{}^2] + \tfrac{1}{2}M_1[L_1{}^2sin^2\theta_1\dot{\theta}_1{}^2] + \tfrac{1}{2}M_2[L_1\ cos\ \theta_1\dot{\theta}_1 + L_2cos(\theta_1+\theta_2)(\dot{\theta}_1+\dot{\theta}_2)\ ]^2$$
$$+ \tfrac{1}{2}M_2[-L_1\ sin\theta_1\ \dot{\theta}_1 - L_2sin(\theta_1+\theta_2)(\dot{\theta}_1+\dot{\theta}_2)\ ]^2$$

$$=\ \tfrac{1}{2}M_1L_1{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2[L_1{}^2c^2\theta_1\dot{\theta}_1{}^2 + L_2{}^2c^2(\theta_1+\theta_2)(\dot{\theta}_1+\dot{\theta}_2)^2$$
$$+ 2L_1L_2c\theta_1c(\theta_1+\theta_2)\dot{\theta}_1(\dot{\theta}_1+\dot{\theta}_2)]\ + \tfrac{1}{2}M_2[L_1{}^2s^2\theta_1\dot{\theta}_1{}^2 + L_2{}^2s^2(\theta_1+\theta_2)(\dot{\theta}_1+\dot{\theta}_2)^2$$
$$+ 2L_1L_2s\theta_1s(\theta_1+\theta_2)\dot{\theta}_1(\dot{\theta}_1+\dot{\theta}_2)]$$

$$= \tfrac{1}{2}M_1L_1{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_1{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2(\dot{\theta}_1+\dot{\theta}_2)^2 + L_1L_2M_2c\theta_1c(\theta_1+\theta_2)\dot{\theta}_1(\dot{\theta}_1+\dot{\theta}_2)$$
$$+ L_1L_2M_2s\theta_1s(\theta_1+\theta_2)\dot{\theta}_1(\dot{\theta}_1+\dot{\theta}_2)$$

$$= \tfrac{1}{2}(M_1+M_2)L_1{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2\dot{\theta}_2{}^2\ + M_2L_2{}^2\dot{\theta}_1\dot{\theta}_2$$
$$+ L_1L_2M_2\dot{\theta}_1(\dot{\theta}_1+\dot{\theta}_2)[c\theta_1c\theta_1c\theta_2 - c\theta_1s\theta_1s\theta_2 + s\theta_1\ s\theta_1\ c\theta_2 + s\theta_1\ c\theta_1\ s\theta_2\ ]$$

After simplification,

$$KE\ = \tfrac{1}{2}(M_1+M_2)L_1{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2\dot{\theta}_2{}^2 + M_2L_2{}^2\dot{\theta}_1\dot{\theta}_2$$
$$+ L_1L_2M_2c\theta_2(\dot{\theta}_1{}^2 + \dot{\theta}_1\dot{\theta}_2)$$

Potential energy can be computed as,

$$PE\ =\ M_1gL_1c\theta_1 + M_2g(L_1c\theta_1 + L_2c(\theta_1+\theta_2)$$

Applying the Euler-Lagrange crank, we can compute the Lagrangian

$$L = KE - PE$$

$$= \tfrac{1}{2}(M_1 + M_2)L_1{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2\dot{\theta}_1{}^2 + \tfrac{1}{2}M_2L_2{}^2\dot{\theta}_2{}^2 + M_2L_2{}^2\dot{\theta}_1\dot{\theta}_2 + L_1L_2M_2c\theta_2(\dot{\theta}_1{}^2 + \dot{\theta}_1\dot{\theta}_2)$$

$$- M_1gL_1c\theta_1 - M_2g(L_1c\theta_1 + L_2c(\theta_1 + \theta_2)$$

Finally, forming the dynamical equations to be,

$$f_{\theta_{1,2}} = \frac{d}{dt}\left[\frac{dL}{d\dot{\theta}_{1,2}}\right] - \frac{dL}{d\theta_{1,2}}$$

Differentiating the terms wrt the joint variable,

$$\frac{d}{dt}\left[\frac{dL}{d\dot{\theta}_{1,2}}\right] = (M_1 + M_2)L_1{}^2\ddot{\theta}_1 + M_2L_2{}^2\ddot{\theta}_1 + M_2L_2{}^2\ddot{\theta}_2 + M_2L_1L_2c\theta_2\ddot{\theta}_2 - L_1L_2M_2c\theta_2(2)\ddot{\theta}_1$$

$$- L_1L_2M_2s\theta_2(2\dot{\theta}_1\dot{\theta}_2)$$

$$\frac{dL}{d\theta_1} = M_1gL_1s\theta_1 + M_2g[L_1s\theta_1] + M_2gL_2s(\theta_1 + \theta_2)$$

After combining and simplifying the equations we get,

$$f(\theta_1) = ((M_1 + M_2)L_1{}^2 + M_2L_2{}^2 + 2M_2L_1L_2c\theta_2)\ddot{\theta}_1 + (M_2L_2{}^2 + M_2L_1L_2c\theta_2)\ddot{\theta}_2$$

$$- (M_1 + M_2)gL_1s\theta_1 - L_1L_2M_2s\theta_2(2\dot{\theta}_1\dot{\theta}_2 + \dot{\theta}_2{}^2) - M_2gL_2s(\theta_1 + \theta_2)$$

Similarly,

$$f(\theta_2) = (M_2L_2{}^2 + M_2L_1L_2c\theta_2)\ddot{\theta}_1 + M_2L_2{}^2\ddot{\theta}_2 - M_2L_2\,L_1s\theta_2\dot{\theta}_1\dot{\theta}_2 - M_2gL_2s(\theta_1 + \theta_2)$$

Now we can describe the motion of the 2 DOF manipulator by

$$D(q)\ddot{q} + C(\dot{q}, q) + g(q) = F$$

q = $[\theta_1 \;\; \theta_2]^T$

$$D(q) = \begin{bmatrix} ((M_1+M_2)L_1^2 + M_2L_2^2 + 2M_2L_2cos\theta_2) & (M_2L_2^2 + M_2L_1L_2cos\theta_2) \\ (M_2L_2^2 + M_2L_1L_2cos\theta_2) & (M_2L_2^2) \end{bmatrix}$$

$$C(\dot{q}, q) = \begin{bmatrix} -M_2L_2L_1sin\theta_2(2\dot{\theta}_1\dot{\theta}_2 + 2\dot{\theta}_2^2) \\ -M_2L_2L_1sin\theta_2(\dot{\theta}_1 + \dot{\theta}_2) \end{bmatrix}$$

$$g(\dot{q}, q) = \begin{bmatrix} -(M_1+M_2)gL_1sin\theta_1 - M_2gL_2sin(\theta_1+\theta_2) \\ -M_2gL_2sin(\theta_1+\theta_2) \end{bmatrix}$$

$$F = [f_{\theta_1} \; f_{\theta_2}]^T$$

B. Control Design

We have system equation

$$D(q)\ddot{q} + C(\dot{q}, q) + g(q) = F$$

We can have

$$\ddot{q} = D(q)^{-1}[-C(\dot{q}, q) - g(q)] + \hat{F}$$

given

$$\hat{F} = D(q)^{-1}F \quad \Leftrightarrow \quad F = D(q)\hat{F}$$

After decoupling the system new input

$$\widehat{F} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

However, the actual input torque is

$$\begin{bmatrix} f_{\theta_1} \\ f_{\theta_2} \end{bmatrix} = B(q) \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

The error signals

$$e(\theta_1) = \theta_{1f} - \theta_1$$
$$e(\theta_2) = \theta_{2f} - \theta_2$$

We assumed initial position

$$\theta_0 = \begin{bmatrix} -\frac{\pi}{2} \\ \frac{\pi}{2} \end{bmatrix}$$

Final position

$$\begin{bmatrix} \theta_{1f} \\ \theta_{2f} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} \\ -\frac{\pi}{2} \end{bmatrix}$$

C. PID Design

General structure of PID controller

$$f = K_P e + K_D \dot{e} + K_I \int \text{e dt}$$

In our case,

$$f_1 = K_{P1}(\theta_{1f} - \theta_1) - K_{D1}\dot{\theta} + K_{I1} \int e(\theta_1)\, dt$$

$$f_2 = K_{P2}(\theta_{2f} - \theta_2) - K_{D2}\dot{\theta} + K_{I2} \int e(\theta_2)\, dt$$

So, complete system equation would be

$$\ddot{q} = B(q)^{-1}[-C(\dot{q}, q) - g(q)] + \hat{F}$$
$$\hat{F} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} K_{P1}(\theta_{1f} - \theta_1) - K_{D1}\dot{\theta_1} + K_{I1} \int e(\theta_1)dt \\ K_{P2}(\theta_{2f} - \theta_2) - K_{D2}\dot{\theta_2} + K_{I2} \int e(\theta_2)dt \end{bmatrix}$$

Upon integration

$$x_1 = \int e(\theta_1)dt \Rightarrow \dot{x}_1 = \theta_{1f} - \theta_1$$

$$x_2 = \int e(\theta_2)dt \Rightarrow \dot{x}_2 = \theta_{2f} - \theta_2$$

Complete system of equations

$$\begin{cases} \dot{x}_1 = \theta_{1f} - \theta_1 \\ \dot{x}_2 = \theta_{2f} - \theta_2 \\ \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} = B(q)^{-1}[-C(\dot{q}, q) - g(q)] + \begin{bmatrix} K_{P1}(\theta_{1f} - \theta_1) - K_{D1}\dot{\theta}_1 + K_{I1}x_1 \\ K_{P2}(\theta_{2f} - \theta_2) - K_{D2}\dot{\theta}_2 + K_{I2}x_2 \end{bmatrix} \end{cases}$$

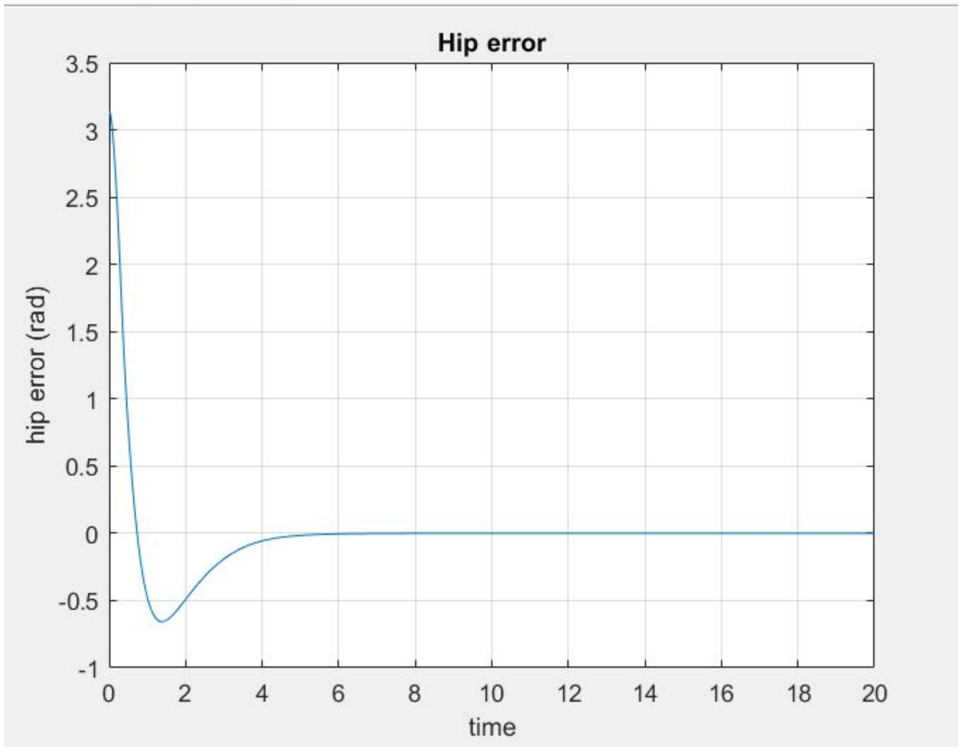
By simulating the model and optimizing the control parameters to have best performance, we obtain the following gain values-

		GWO	PSO
PID1	Kp	1.8	15
	Ki	-0.09	7
	Kd	1.34	10
PID2	Kp	0.88	15
	Ki	0.27	10
	Kd	1.079	10

Table 1: PID Gains

D. State Results

Error graphs for $\theta_1(hip)$ and $\theta_2(knee)$ is shown below-



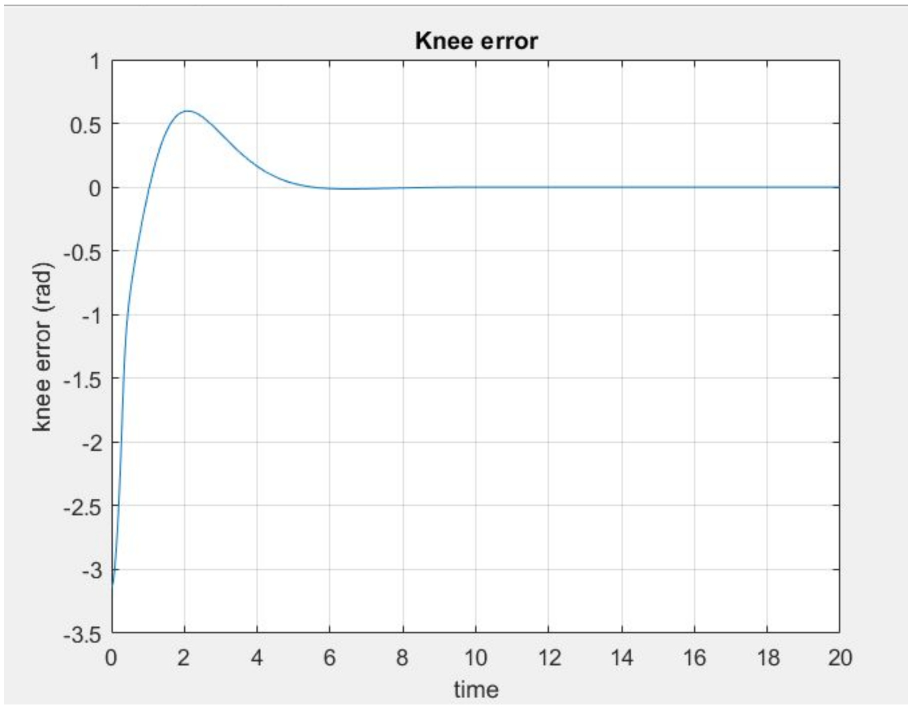


Figure 3: Error graphs for $\theta_1(hip)$ and $\theta_2(knee)$

From the two error graphs shown above it can be concluded that after some time the error associated with the two joints tends to zero thereby stabilizing the system.

E. Torque Results

Here we analyze the torques obtained. The inputs for the system are the two joint torques.

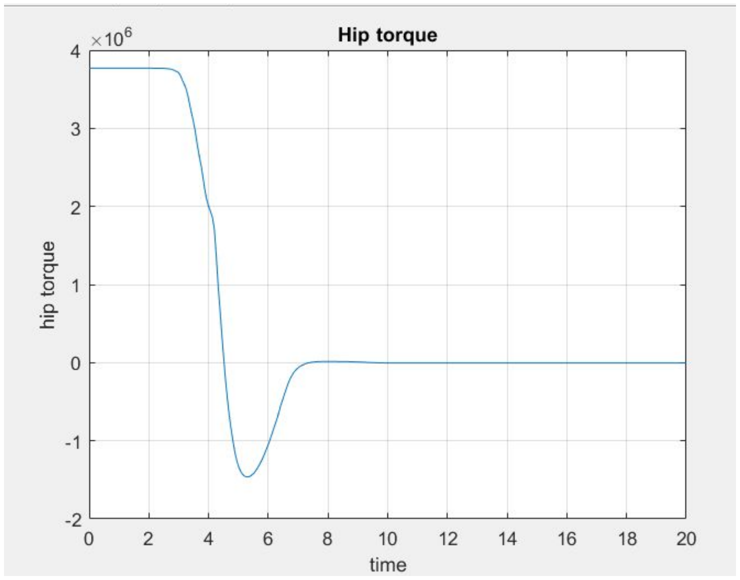


Figure 4: Torque graph for $\theta_1(hip)$

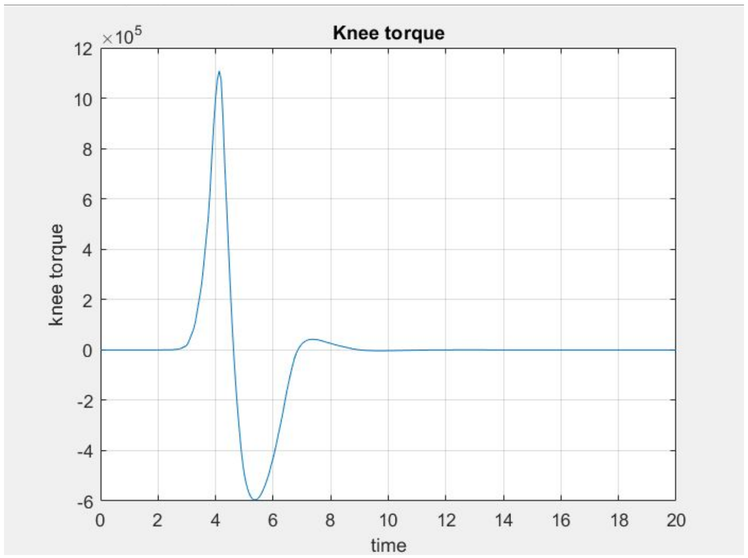


Figure 5: Torque graph $\theta_2(knee)$

Optimization algorithms

We used 2 optimization algorithms.

1. Particle swarm optimization
2. Grey wolf optimization

Particle swarm optimization (PSO)[6]

Particle swarm optimization is a computation method used to optimize a problem using iterative approach. PSO is inspired by the behaviour of birds or fishes. It solves problem by generating initial population of particle randomly. A particle is just a point in n dimensional hyperspace, where n is the number of input variables. Then iteratively calculate the cost function for each particle. After each iteration cost for each particle is calculated and a slight nudge is given to each particle. This is done based on the position of the particle with the lost cost function. The goal is to find the global minimum of the n dimensional hyperspace created by the cost function.

Grey wolf optimization (GWO)[5]

Grey wolf optimization is similar technique as particle swarm optimization. It generates an initial random population then compute the cost for each individual. Based on this population is divided into four categories- alpha, beta, delta and omega.

Grey wolf optimization intends to mimic the hunting strategy of wolves, alpha is authorized to take decisions. Beta is the second is hierarchy and delta is last. All other which do not belong to these categories belong to omega. Omega are the weakest.

Limitations of these optimization methods

Both of the optimization algorithm being used are trying to find the global minimum of the cost function. But there is a possibility of the optimizer getting stuck in one of two regions- local minimum or plateau region.

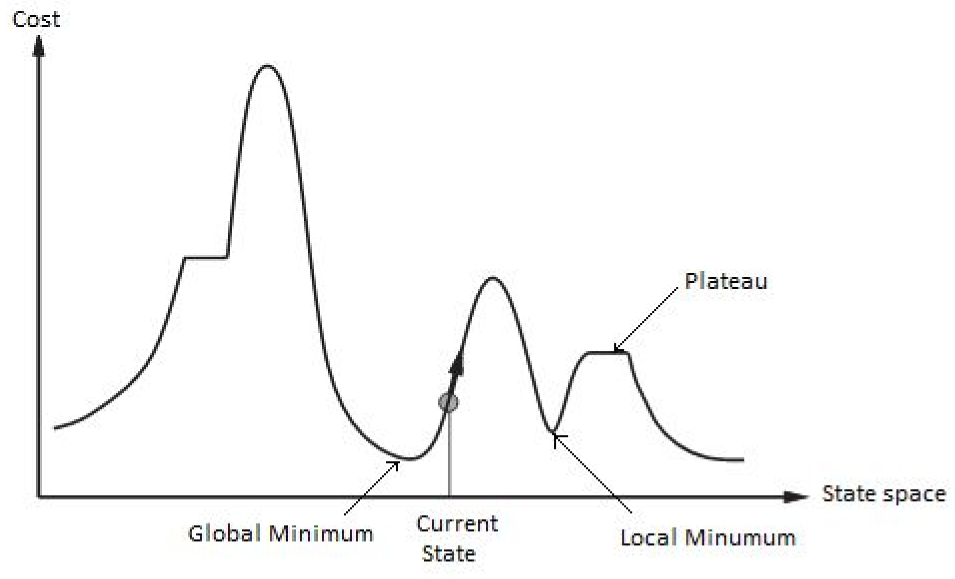


Figure 6: State space vs cost mapping

Local minimum

Local minimum is not the least value that possible for a given cost function, but the cost function increases if any dimension is changed.

Plateau region

Plateau region is area in hyperspace where change in any parameter results in the same value of cost function.

Results

The following table shows the parameters that were tuned by the optimization algorithm.

		GWO	PSO
PID1	Kp	1.8	15
	Ki	-0.09	7
	Kd	1.34	10
PID2	Kp	0.88	15
	Ki	0.27	10
	Kd	1.079	10

Table 2: Gains obtained from the optimization algorithms

Cost per iteration

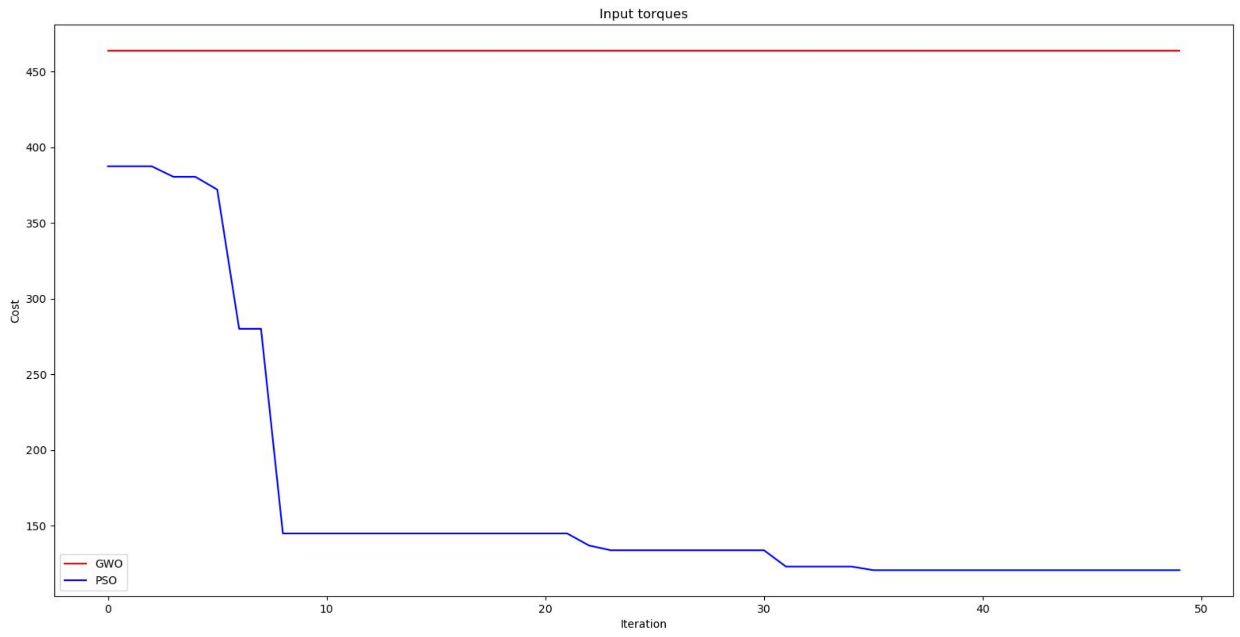


Figure 7: Cost per iteration for GWO and PSO

Thus, from the results we found that PSO gave the better cost and we then further implemented the system designed above in MATLAB and used the PID parameters obtained from PSO. Initial states for hip and knee are assumed as -pi/2 and pi/2 respectively and final position for hip and knee joints are assumed as pi/2 and -pi/2.

LQR

The Linear Quadratic Regulator (LQR) is a well-known method that provides optimally controlled feedback gains to enable the closed-loop stable and high-performance design of systems.

For the derivation of the linear quadratic regulator we consider a linear system state-space representation:

$$x' = Ax + Bu$$
$$y' = Cx$$

For this continuous-time linear system, defined with a quadratic cost function defined as:

$$J = x^T(t_1)F(t_1)x(t_1) + \int_{t_0}^{t_1} \left(x^T Q x + u^T R u + 2x^T N u\right) dt$$

the feedback control law that minimizes the value of the cost is:

$u = -Kx$ where K is given by:

$K = R^{-1}(B^T P(t) + N^T)$ and P is found by solving the Riccati differential equation.

We need to implement LQR controller for the 2 DOF manipulator which is a time varying nonlinear system. So, in order to design the controller and model the system in state space form we will first need to linearize the system.

Linearization

To linearize the system, we assume that the system is operating about a nominal state $(\theta_1 = \theta_2 = 0^o)$

The linearized equations are then developed using the Taylor series expansion and we write the actual state as $x(t) = x_e + \delta x(t)$ and the actual input as $u(t) \ = u_e + \delta u(t)$.

Now combining all the n state equations yields:

$$\frac{d}{dt}\delta\mathbf{x} = \begin{bmatrix} \left.\frac{\partial f_1}{\partial \mathbf{x}}\right|_0 \\ \left.\frac{\partial f_2}{\partial \mathbf{x}}\right|_0 \\ \vdots \\ \left.\frac{\partial f_n}{\partial \mathbf{x}}\right|_0 \end{bmatrix} \delta\mathbf{x} + \begin{bmatrix} \left.\frac{\partial f_1}{\partial \mathbf{u}}\right|_0 \\ \left.\frac{\partial f_2}{\partial \mathbf{u}}\right|_0 \\ \vdots \\ \left.\frac{\partial f_n}{\partial \mathbf{u}}\right|_0 \end{bmatrix} \delta\mathbf{u}$$

$$= A(t)\delta\mathbf{x} + B(t)\delta\mathbf{u}$$

where

$$A(t) \equiv \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ & & \vdots & \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}_0 \quad \text{and} \quad B(t) \equiv \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_m} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \cdots & \frac{\partial f_2}{\partial u_m} \\ & & \vdots & \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \cdots & \frac{\partial f_n}{\partial u_m} \end{bmatrix}_0$$

State space Model

We choose the four states for system and then linearize the dynamical equations to obtain the linear state space model of the system described above in figure 2. We assumed the value of M_1, M_2, L_1, L_2 all as 1.

$$x_1 = \theta_1$$
$$x_2 = \dot{\theta}_1$$
$$x_3 = \theta_2$$
$$x_4 = \dot{\theta}_2$$

$$\begin{bmatrix} \dot{\theta}_1 \\ \ddot{\theta}_1 \\ \dot{\theta}_2 \\ \ddot{\theta}_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0.4 & 0.2 & 0 & -0.2 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & -2 & 0 \end{bmatrix} * \begin{bmatrix} \theta_1 \\ \dot{\theta}_1 \\ \theta_2 \\ \dot{\theta}_1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0.33 & -0.33 \\ 0 & 0 \\ -0.67 & 1.67 \end{bmatrix} * \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}$$

$$y = [\theta_1 \ \ \theta_2]^T$$

$$\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} \theta_1 \\ \dot{\theta}_1 \\ \theta_2 \\ \dot{\theta}_1 \end{bmatrix}$$

After obtaining the linearized state space model we simulated the system in MATLAB with different state (Q) and control (R) weighting matrices. The eigenvalues computed before applying the LQR controller existed in the right half plane which rendered the system unstable. But after applying the LQR feedback control to the system we were able to render it stable as the new eigenvalues computed were in the open left half plane.

```
>> LQR

EigenValueBefore =

   0.0459 + 1.4691i
   0.0459 - 1.4691i
   0.6651 + 0.0000i
  -0.5568 + 0.0000i


EigenValueAfter =

  -1.5494 + 1.1818i
  -1.5494 - 1.1818i
  -0.5535 + 0.3005i
  -0.5535 - 0.3005i
```

Conclusion

The main aim of this study was to develop an optimal controller which consists of a combination of LQR and PID controller for quadruped robots by optimizing it with GWO. In the paper “Grey Wolf Optimizer Based Tuning of a Hybrid LQR-PID Controller for Foot Trajectory Control of a Quadruped Robot” authors concluded that Grey wolf optimization is the best way to find out optimal values of PID and LQR control with the assumption that it is better to use optimization algorithm to find these values instead of trial and error. But we concluded that these optimization algorithms also have hyperparameters of their own to get optimal results and PSO performed better than GWO.

Bibliography

[1]https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-30-feedback-control-systems-fall-2010/lecture-notes/MIT16_30F10_lec05.pdff

[2] http://www.kostasalexis.com/lqr-control.html

[3] https://en.wikipedia.org/wiki/Linear%E2%80%93quadratic_regulator

[4]https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec20.pdf

[5]https://www.sciencedirect.com/science/article/pii/S0965997813001853

[6]https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=488968&tag=1

Appendix

Code

robot.m

```
function xDot = robot(t,x,thFinal,specification,kPID)
xDot = zeros(8,1);
th1Final = thFinal(1);
th2Final = thFinal(2);
%% Robot Specifications
m1 = specification(3);
m2 = specification(4);
l1 = specification(1);
l2 = specification(2);
g = 9.8;
%% Inertia Matrix
b11 = (m1 + m2) * l1^2 + m2 * l2^2 + 2 * m2 * l1 * l2 *
cos(x(4));
b12 = m2 * l2^2 + m2 * l1 * l2 * cos(x(4));
b21 = m2 * l2^2 + m2 * l1 * l2 * cos(x(4));
b22 = m2 * l2^2;
Bq = [b11 b12; b21 b22];
%% C Matrix
c1 = -m2 * l1 * l2 * sin(x(4)) * (2 * x(5) * x(6) +
x(6)^2);
c2 = -m2 * l1 * l2 * sin(x(4)) * x(5) * x(6);
Cq = [c1; c2];
%% Gravity Matrix
g1 = -(m1 + m2) * g * l1 * sin(x(3)) - m2 * g * l2 *
sin(x(3) + x(4));
g2 = -m2 * g * l2 * sin(x(3) + x(4));
Gq = [g1; g2];
```

```
%% PID Control
kP1 = kPID(1);
kD1 = kPID(2);
kI1 = kPID(3);
kP2 = kPID(4);
kD2 = kPID(5);
kI2 = kPID(6);
f1 = kP1 * (th1Final - x(3)) - kD1 * x(5) + kI1 * (x(1));
f2 = kP2 * (th2Final - x(4)) - kD2 * x(6) + kI2 * (x(2));
fHat = [f1; f2];
F = Bq * fHat;
%% System states
xDot(1) = (th1Final - x(3));
xDot(2) = (th2Final - x(4));
xDot(3) = x(5);
xDot(4) = x(6);
q2dot = inv(Bq) * (-Cq -Gq +F);
xDot(5) = q2dot(1);
xDot(6) = q2dot(2);
xDot(7) = F(1);
xDot(8) = F(2);
```

control.m

```
clear
clc
%% Initilization
thInt = [-pi/2 pi/2]; % starting positions
thFinal = [pi/2 -pi/2]; % final position
x0 = [0 0 thInt 0 0 0 0];
time = [0 20]; % time
%% Robot Properties
l1 = 200; %link 1
```

```
l2 = 200; %link 2
m1 = 1; %mass 1
m2 = 1; %mass 2
specification = [l1 l2 m1 m2];
%% PID Parameters
kP1 = 15;
kD1 = 7;
kI1 = 10;
kP2 = 15;
kD2 = 10;
kI2 = 10;
kPID = [kP1 kD1 kI1 kP2 kD2 kI2];
%% ODE
[Y, X] = ode45(@(y, x) robot(y, x, thFinal, specification,
kPID), time, x0);
%% Output
theta1 = X(:, 3);
theta2 = X(:, 4);
% torque
f1 = diff(X(: ,7))./diff(Y);
f2 = diff(X(:, 8))./diff(Y);
t = 0:(Y(end) / (length(f1) - 1)):Y(end);
% X - Y
x1 = l1.*sin(theta1); % X1
y1 = l1.*cos(theta1); % Y1
x2 = l1.*sin(theta1)+l2.*sin(theta1+theta2); % X2
y2 = l1.*cos(theta1)+l2.*cos(theta1+theta2); % Y2
% theta1 plot for error
plot(Y, thFinal(1) - theta1)
grid
title('Hip error')
ylabel('hip error (rad)')
```

```
xlabel('time')
%theta2 error plot
figure
plot(Y, thFinal(2) - theta2)
grid
title('Knee error')
ylabel('knee error (rad)')
xlabel('time')
%torque1 plot
figure
plot(t, f1)
grid
title('Hip torque')
ylabel('hip torque')
xlabel('time')
%torque2 plot
figure
plot(t, f2)
grid
title('Knee torque')
ylabel('knee torque')
xlabel('time')
```

LQR.m

```
A = [0 1 0 0; 0.4 0.2 0 -0.2; 0 0 0 1; 1 1 -2 0];
B = [0 0; 0.33 -0.33; 0 0; -0.67 0.67];
Q = [12.22 0 0 0; 0 0.99 0 0; 0 0 0.78 0; 0 0 0 45.74];
R = 6.88;

K = lqr(A, B, Q, R);

EigenValueBefore  = eig(A)
```

```
EigenValueAfter = eig(A-B*K)
```

Simulator.py

```python
import pymunk
from numpy import pi


# print(tor1, tor2)
space = pymunk.Space()
space.gravity = float(0), float(0)

ground = pymunk.Body(body_type = pymunk.Body.STATIC)
ground.position = (float(320), float(-5))
polyG =pymunk.Poly.create_box(ground,  size=(float(640), float(10)))

rotation_center_body = pymunk.Body(body_type = pymunk.Body.STATIC)
rotation_center_body.position = (float(320), float(285.84))

poly = pymunk.Poly.create_box(None, size=(float(10), float(200)))
moment = pymunk.moment_for_poly(float(1), poly.get_vertices())
body = pymunk.Body(float(1),float(moment))
poly.body = body
body.position = float(249.29), float(215.13)
body.angle = float(-(45 * pi /180))

poly2 = pymunk.Poly.create_box(None, size=(float(10), float(200)))
moment2 = pymunk.moment_for_poly(float(1), poly2.get_vertices())
body2 = pymunk.Body(float(1),float(moment2))
poly2.body = body2
body2.position = float(249.29), float(73.71)
body2.angle = float(45 * pi / 180)

rotation_center_joint = pymunk.PinJoint(body, rotation_center_body,
(float(0),float(100)), (float(0),float(0)))
rotation_center_joint2 = pymunk.PinJoint(body, body2, (float(0),float(-100)),
(float(0),float(105)))

space.add(body,body2, poly, poly2, rotation_center_joint, rotation_center_joint2)

def reset():
   body.angle = float(-(45 * pi / 180))
   body2.angle =float(45 * pi / 180)
   space.step(0.01)
```

```
def Update(tor1, tor2):
   tor1 = tor1 * 100000
   tor2 = tor2 * 100000
   # if tor1 > 100000000:
   #     tor1 = 100000000
   # if tor2 > 100000000:
   #     tor2 = 100000000
   # print("tor", tor1, tor2)

   body.torque = float(tor1)
   body2.torque = float(tor2)
   space.step(0.01)
   # print(body.angle, body2.angle)
   return body.angle, body2.angle
```

Graphs.py

```
def plotPerformanceGraph(plt, xRef, yRef, xGWO, yGWO, xPSO, yPSO, xGA, yGA):
   plt.subplot(2, 2, 1)
   plt.plot(xRef, yRef, color='black', label='reference')
   plt.plot(xGWO, yGWO, color='red', label='GWO')
   plt.plot(xPSO, yPSO, color='blue', label='PSO')
   # plt.plot(xGA, yGA, color='green', label='GA')
   plt.xlabel('x - axis')
   plt.ylabel('y - axis')
   plt.title('Trajectory control')
   plt.legend()

def plotPoseGraph(plt, xGWO, yGWO, xPSO, yPSO, xGA, yGA):
   plt.subplot(2, 2, 2)
   plt.plot(xGWO, yGWO, color='red', label='GWO')
   plt.plot(xPSO, yPSO, color='blue', label='PSO')
   # plt.plot(xGA, yGA, color='green', label='GA')
   plt.xlabel('Time')
   plt.ylabel('Trajectory Tracking error')
   plt.title('Absolute trajectory errors ')
   plt.legend()

def plotTorqueGraph(plt, xGWO, yGWO, xPSO, yPSO, xGA, yGA):
   plt.subplot(2, 2, 3)
   plt.plot(xGWO, yGWO, color='red', label='GWO')
   plt.plot(xPSO, yPSO, color='blue', label='PSO')
   # plt.plot(xGA, yGA, color='green', label='GA')
   plt.xlabel('Time')
   plt.ylabel('Torque')
   plt.title('Input torques')
   plt.legend()
```

```python
def plotCostGraph(plt, xGWO, yGWO, xPSO, yPSO, xGA, yGA):
   # plt.subplot(2, 2, 4)
   plt.plot(xGWO, yGWO, color='red', label='GWO')
   plt.plot(xPSO, yPSO, color='blue', label='PSO')
   # plt.plot(xGA, yGA, color='green', label='GA')
   plt.xlabel('Iteration')
   plt.ylabel('Cost')
   plt.title('Input torques')
   plt.legend()
```

GWO1.py

```python
import random
import numpy
import solution as s
import time
import test

def GWO(objf, lb, ub, dim, SearchAgents_no, Max_iter):
   # Max_iter=1000
   # lb=-100
   # ub=100
   # dim=30
   # SearchAgents_no=5

   # initialize alpha, beta, and delta_pos
   Alpha_pos = numpy.zeros(dim)
   Alpha_score = float("inf")

   Beta_pos = numpy.zeros(dim)
   Beta_score = float("inf")

   Delta_pos = numpy.zeros(dim)
   Delta_score = float("inf")

   if not isinstance(lb, list):
       lb = [lb] * dim
   if not isinstance(ub, list):
       ub = [ub] * dim

   # Initialize the positions of search agents
   Positions = numpy.zeros((SearchAgents_no, dim))
   for i in range(dim):
       Positions[:, i] = numpy.random.uniform(0, 1, SearchAgents_no) * (ub[i] -
lb[i]) + lb[i]

   Convergence_curve = numpy.zeros(Max_iter)
```

```
    # Loop counter
    print("GWO is optimizing  \"" + objf.__name__ + "\"")

    timerStart = time.time()
    s.startTime = time.strftime("%Y-%m-%d-%H-%M-%S")
    # Main loop
    for l in range(0, Max_iter):
        for i in range(0, SearchAgents_no):

            # Return back the search agents that go beyond the boundaries of the
search space
            for j in range(dim):
                Positions[i, j] = numpy.clip(Positions[i, j], lb[j], ub[j])

            # Calculate objective function for each search agent
            fitness = objf(Positions[i, :])

            # Update Alpha, Beta, and Delta
            if fitness < Alpha_score:
                Alpha_score = fitness;  # Update alpha
                Alpha_pos = Positions[i, :].copy()

            if (fitness > Alpha_score and fitness < Beta_score):
                Beta_score = fitness  # Update beta
                Beta_pos = Positions[i, :].copy()

            if (fitness > Alpha_score and fitness > Beta_score and fitness <
Delta_score):
                Delta_score = fitness  # Update delta
                Delta_pos = Positions[i, :].copy()

        a = 2 - l * ((2) / Max_iter);  # a decreases linearly fron 2 to 0

        # Update the Position of search agents including omegas
        for i in range(0, SearchAgents_no):
            for j in range(0, dim):
                r1 = random.random()  # r1 is a random number in [0,1]
                r2 = random.random()  # r2 is a random number in [0,1]

                A1 = 2 * a * r1 - a;  # Equation (3.3)
                C1 = 2 * r2;  # Equation (3.4)

                D_alpha = abs(C1 * Alpha_pos[j] - Positions[i, j]);  # Equation
(3.5)-part 1
                X1 = Alpha_pos[j] - A1 * D_alpha;  # Equation (3.6)-part 1

                r1 = random.random()
                r2 = random.random()

                A2 = 2 * a * r1 - a;  # Equation (3.3)
                C2 = 2 * r2;  # Equation (3.4)

                D_beta = abs(C2 * Beta_pos[j] - Positions[i, j]);  # Equation
(3.5)-part 2
                X2 = Beta_pos[j] - A2 * D_beta;  # Equation (3.6)-part 2

                r1 = random.random()
                r2 = random.random()

                A3 = 2 * a * r1 - a;  # Equation (3.3)
                C3 = 2 * r2;  # Equation (3.4)

                D_delta = abs(C3 * Delta_pos[j] - Positions[i, j]);  # Equation
(3.5)-part 3
                X3 = Delta_pos[j] - A3 * D_delta;  # Equation (3.5)-part 3

                Positions[i, j] = (X1 + X2 + X3) / 3  # Equation (3.7)

        Convergence_curve[l] = Alpha_score;

        if (l % 1 == 0):
            print(['At iteration ' + str(l) + ' the best fitness is ' +
str(Alpha_score)]);

   timerEnd = time.time()
   s.alpha = Alpha_pos
   s.endTime = time.strftime("%Y-%m-%d-%H-%M-%S")
   s.executionTime = timerEnd - timerStart
   s.convergence = Convergence_curve
   s.optimizer = "GWO"
   s.objfname = objf.__name__

   return  Convergence_curve, Alpha_pos

def runGWO(itr):
   return  GWO(test.testgwo,-40, 40, 3, 10, itr)

# runGWO(50)

PSO.py
import pyswarms as ps
import test

def costFunction(x):
   j = []
   for i in x:
       j.append(test.test(i[0], i[1], i[2]))
   return j
def runPSO(val):
   options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}

   optimizer = ps.single.GlobalBestPSO(n_particles=10, dimensions=3, options=options)
   cost, pos = optimizer.optimize(costFunction, iters=val)
   return optimizer.cost_history, pos
```

Demo.py

```
import graphs
import matplotlib.pyplot as plt
from numpy import arange
from numpy import pi
from numpy import sin
from numpy import cos
import itertools
import Simulator
from math import acos, atan, sqrt
import PSO
import gademo
import GWO1
import test

itr = 50
def getPositionFromSimulation(x, y):
    th1, th2 = Simulator.Update(x, y)
    xcor = 200 * sin(th1) + 200 * sin(th2)
    ycor = 200 * cos(th1) + 200 * cos(th2)
    ycor = ycor - 282.84
    return xcor, ycor


performanceXGWO, performanceYGWO, performanceXPSO, performanceYPSO, performanceXGA,
performanceYGA = [], [], [], [], [], []
poseXGWO, poseYGWO, poseXPSO, poseYPSO, poseXGA, poseYGA = [], [], [], [], [], []
torqueXGWO, torqueYGWO, torqueXPSO, torqueYPSO, torqueXGA, torqueYGA = [], [], [],
[], [], []
costXGWO, costYGWO, costXPSO, costYPSO, costXGA, costYGA = [], [], [], [], [], []

t = arange(-pi / 2, pi / 2, 0.01)
r = 100
xRef = r * sin(t) + 100
yRef = r * cos(t)
xRef = list(xRef)
yRef = list(yRef)
```

```python
xn = list(range(200, 0, -1))
yn = [0] * 200
xRef.extend(xn)
yRef.extend(yn)

xReference = list(itertools.chain.from_iterable(itertools.repeat(x, 10) for x in
xRef))
yReference = list(itertools.chain.from_iterable(itertools.repeat(x, 10) for x in
yRef))
thRef1 = []
thRef2 = []

for xref, yref in zip(xReference, yReference):
   distance = sqrt((xref - 0) * (xref - 0) + (yref - 282.84) * (yref - 282.84))

   thref2 = (pi) - acos(((200 * 200) + (200 * 200) - (distance * distance)) / (2 *
200 * 200))
   thref1 = atan((0 - xref) / (yref - 282.84))

   thref1 = thref1 - acos(((200 * 200) - (200 * 200) + (distance * distance)) / (2 *
200 * distance))
   thRef1.append(thref1)
   thRef2.append(thref2)

yCostPSO, KpidPSO = PSO.runPSO(itr)
yCostGWO, KpidGWO = GWO1.runGWO(itr)

# yCostGA, KpidGA = gademo.runGA(itr)
# yCostGA = [1000-x for x in yCostGA]

trackingErrorPSO = test.resultPlot(KpidPSO[0], KpidPSO[1], KpidPSO[2])
trackingErrorGWO = test.resultPlot(KpidGWO[0], KpidGWO[1], KpidGWO[2])
print(trackingErrorGWO)


# graphs.plotPerformanceGraph(plt, xRef, yRef, list(range(1,5)), list(range(1,5)),
list(range(1,5)), list(range(1,5)), list(range(1,5)), list(range(1,5)))
# graphs.plotPoseGraph(plt, list(range(len(trackingErrorGWO))), trackingErrorGWO,
list(range(len(trackingErrorPSO))), trackingErrorGWO, list(range(5, 10)),
list(range(5, 10)))
# graphs.plotTorqueGraph(plt, list(range(10, 15)), list(range(10, 15)),
list(range(10, 15)), list(range(10, 15)), list(range(10, 15)), list(range(10, 15)))
graphs.plotCostGraph(plt, list(range(itr)), yCostGWO, list(range(itr)), yCostPSO,
list(range(itr)), list(range(itr)))

plt.show()
```