

STATE UNIVERSITY OF NEW YORK AT BUFFALO
Mechanical and Aerospace Engineering Department.

MAE 609 HIGH PERFORMANCE COMPUTING

Final Project

**Distributed Motion Planning for Robot Collectives within
an Artificial Potential Field using Penalty-Based
Approach**

NAME: LENG-FENG LEE

DATE: 19th Dec 2005.

**DECENTRALIZED MOTION PLANNING WITHIN AN ARTIFICIAL POTENTIAL FIELD
APPROACH FOR ROBOT COLLECTIVES WITH PENALTY-BASED FORMULATION**

MAE609 High Performance Computing I

Leng-Feng Lee, Advisor: Dr. Patra & Dr. Jones.

ABSTRACT: In this project, the motion planning of a three point-mass robot collectives is formulated as a holonomic constrained dynamical system. The constrained equations can then be solved using penalty method, in which the holonomic constraints are approximated using spring and dampers. This allows the decentralized formulation of the mechanical system and thus allows us to run the simulation in a distributed manner. Previous attempt has successfully simulated this motion planning in a centralized manner. In this project, we wish to show the ability of this algorithm in a distributed manner by using a formation of 3 robots in two case studies: (1) a quadratic potential field without obstacles, and (2) a navigation function potential field with one obstacle.

INTRODUCTION:

Part A: Constrained-Mechanical System.

The arrangement of a three point-mass robot collectives arranged in a triangular formation in the world fixed frame is shown in Figure 1.

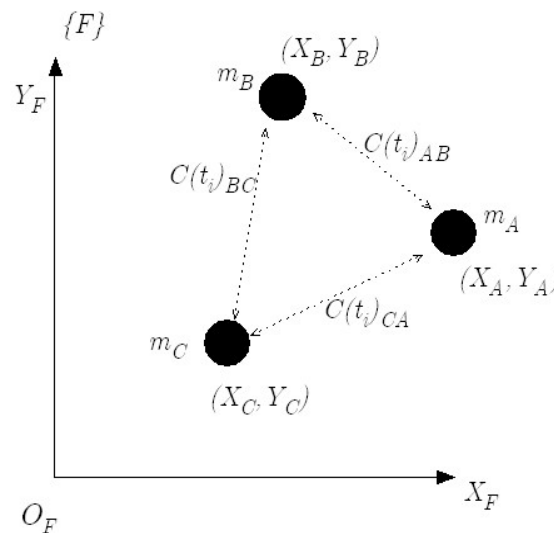


Figure 1: Schematic diagram of a three point-mass robot collectives arrange in a triangular formation.

The dynamics of group or such robot collectives can be formulated as Lagrangian equations of the first kind as:

$$\dot{\mathbf{q}} = \mathbf{v} \quad (1)$$

$$\mathbf{M}(\mathbf{q})\dot{\mathbf{v}} = \mathbf{f}(\mathbf{q}, \mathbf{v}, t, \mathbf{u}) - \mathbf{J}(\mathbf{q})^T \boldsymbol{\lambda} \quad (2)$$

$$\mathbf{C}(\mathbf{q}, t) = \mathbf{0} \quad (3)$$

where \mathbf{q} is the n -dimensional vector of generalized coordinates; \mathbf{v} is the n -dimensional vector of generalized velocities; $\mathbf{M}(\mathbf{q})$ is the $2n \times 2n$ dimensional inertia matrix; $\mathbf{f}(\mathbf{q}, \mathbf{v}, t, \mathbf{u})$ is the n -dimensional vector of external forces; \mathbf{u} is the vector of input forces, which is $-\mathbf{k}_f \nabla_{\mathbf{q}} U$, where U is the potential of the workspace; $\mathbf{C}(\mathbf{q}, t)$ is a m -dimensional vector of holonomic constraints which may or may not depends on time, t ; and $\mathbf{J}(\mathbf{q}) = \frac{\partial \mathbf{C}(\mathbf{q})}{\partial \mathbf{q}}$ is the *Jacobian* matrix. These coupled equations can be solved using the converted ODE approach, where all the algebraic position and velocity level constraints are differentiated and represented at the acceleration level to obtain an augmented index-1 DAE, in terms of the unknown accelerations and the unknown multipliers.

While the converted ODE can be solve using various method [], the penalty based method is that adopted in this project is the widely use one since it provide a decentralized formulation of the dynamic equations.

In this method, the $\boldsymbol{\lambda}$ in Equation (2) is replaced by the constraints expressed as:

$$\boldsymbol{\lambda} = \mathbf{K}_s \mathbf{C}(\mathbf{q}, t) + \mathbf{K}_d \dot{\mathbf{C}}(\mathbf{q}, t) \quad (4)$$

The approach can be visualized in Figure 2, where the holonomic constraints are replaced by using virtual spring and dampers to ensure the holonomic constraints are satisfied.

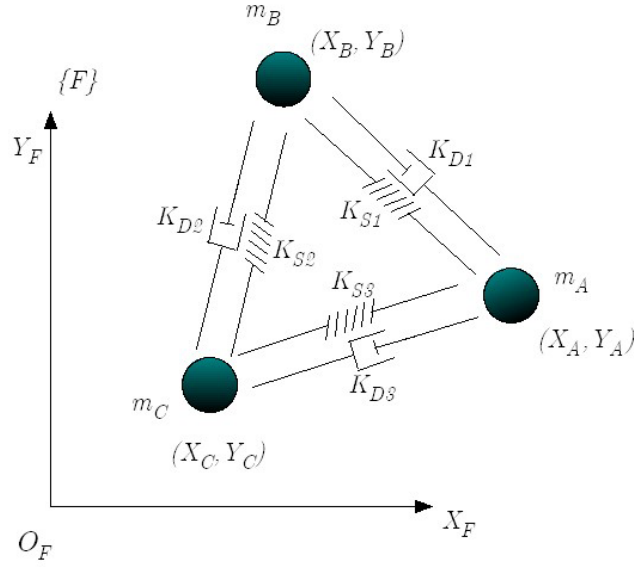


Figure 2: In penalty formulation, the Lagrangian multiplier is approximated using virtual springs and damper to ensure the holonomic constraints are satisfied.

Using this method, it can be shown that the centralized formulation of the penalty formulation approach can be written in state-space form as:

$$\begin{bmatrix} \dot{\mathbf{q}}_{2n \times 1} \\ \ddot{\mathbf{q}}_{2n \times 1} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{M}^{-1} [\mathbf{u} - \mathbf{K}\dot{\mathbf{q}} - \mathbf{J}^T(\mathbf{q})(\mathbf{K}_S \mathbf{C}(\mathbf{q}) + \mathbf{K}_D \dot{\mathbf{C}}(\mathbf{q}))] \end{bmatrix} \quad (5)$$

Note that the state vector $\mathbf{q} = [\mathbf{q}_A^T, \mathbf{q}_B^T, \mathbf{q}_C^T]^T$ has the state variables that belong to robot A, B, and C. The distributed model of each of respective robot may be obtained in the state-space form as

$$\begin{aligned} \begin{bmatrix} \dot{\mathbf{q}}_A \\ \ddot{\mathbf{q}}_A \end{bmatrix}_{4 \times 1} &= \begin{bmatrix} \mathbf{v}_A \\ \mathbf{M}_A^{-1} [\mathbf{E}_A \mathbf{u}_A - \mathbf{K}_A \dot{\mathbf{q}}_A - \mathbf{J}_A^T (\mathbf{K}_{S,A} \mathbf{C} + \mathbf{K}_{D,A} \dot{\mathbf{C}}_A)] \end{bmatrix} \\ \begin{bmatrix} \dot{\mathbf{q}}_B \\ \ddot{\mathbf{q}}_B \end{bmatrix}_{4 \times 1} &= \begin{bmatrix} \mathbf{v}_B \\ \mathbf{M}_B^{-1} [\mathbf{E}_B \mathbf{u}_B - \mathbf{K}_B \dot{\mathbf{q}}_B - \mathbf{J}_B^T (\mathbf{K}_{S,B} \mathbf{C} + \mathbf{K}_{D,B} \dot{\mathbf{C}}_B)] \end{bmatrix} \\ \begin{bmatrix} \dot{\mathbf{q}}_C \\ \ddot{\mathbf{q}}_C \end{bmatrix}_{4 \times 1} &= \begin{bmatrix} \mathbf{v}_C \\ \mathbf{M}_C^{-1} [\mathbf{E}_C \mathbf{u}_C - \mathbf{K}_C \dot{\mathbf{q}}_C - \mathbf{J}_C^T (\mathbf{K}_{S,C} \mathbf{C} + \mathbf{K}_{D,C} \dot{\mathbf{C}}_C)] \end{bmatrix} \end{aligned} \quad (6)$$

where $\mathbf{u}_i = \nabla_{\mathbf{q}_i} U$, $\dot{\mathbf{C}}_i = [\mathbf{J}_i][\dot{\mathbf{q}}_i]$, and $\mathbf{K}_{S,i}$, $\mathbf{K}_{D,i}$ with $i = A, B, C$ are the compliance matrices for springs and dampers respectively.

The three dynamic sub-systems shown in Equation (6), can be simulated in a distributed manner if at every time step: (1) either the information pertaining to $\mathbf{C}(\mathbf{q})$, the extend of the constraint violation, is made available explicitly or (2) computed by exchanging information between the robots. The sole coupling between the three sub-parts is due to the Lagrange multipliers, which are now explicitly calculated using the virtual spring. Equations in (6) are second-order ODEs that need to be solved numerically. In particular, a fixed time-step solver is desired since in physical implementation of the algorithm, data are transmitted in a fixed time interval.

Part B: Artificial Potential Field Approach

The artificial potential field approach is a widely used approach in the robotic field for motion planning. It is a explicit method in which the path of the robot is not known a priori. The fundamental idea behind potential field approaches is to treat the target position as a attractive well, where the minimum is at the target; and to treat obstacles as high potential hill that create a repulsive force. Superimpose the attractive potential and repulsive potential gives you the total artificial potential of the work space.

To ensure that the robot will reach its target, it thus required that the potential of the workspace has a unique minimum and is smooth else where: meaning there are no local minima in the workspace. While many researchers tried to create such artificial potentials (for example harmonic potential field, FIRAS function, superquadric potential, etc), only the navigation function are able to guarantee a unique minima in the workspace.

In short, the navigation function of a ‘sphere world’ is given by the following equation:

$$\varphi_{\kappa}(\mathbf{q}) = U_{Total}(\mathbf{q}) = \begin{cases} \left(\rho_{\kappa} \circ \sigma_1 \circ \frac{\gamma_{\kappa}}{\beta} \right) = \frac{\|\mathbf{q} - \mathbf{q}_{Tar}\|^2}{\left[\|\mathbf{q} - \mathbf{q}_{Tar}\|^{2\kappa} + \beta(\mathbf{q}) \right]^{1/\kappa}} & \text{for } \beta > 0 \\ 1 & \text{for } \beta \leq 0 \end{cases} \quad (7)$$

In particular, the κ value controls the shape of the potential field. It is a tunable parameter that needs to be select such that a smooth potential field can be obtained.

In potential field approach, the gradient information of a potential field is the direction of the input force to the system. Thus finding the gradient of the potential is

necessary. Some simple shaped potential fields allow us to obtain the gradient information analytically. In the case of the navigation function given as Equation (7), the gradient is obtained numerically. The 3 dimensional and contour plot of the potential field created using the navigation function used in our case study were shown in Figure 3 (a) and (b).

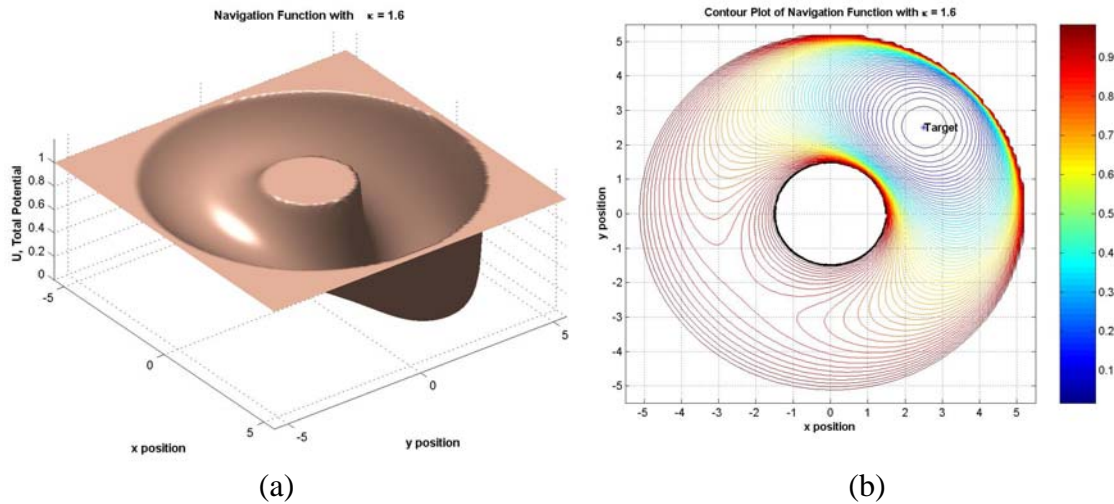


Figure 3: The (a) 3D and contour plot of the potential field used in the case study.

PARALLELIZATION APPROACHES:

The ODE equations shown in Equation (6) need to solve numerically. Each processor represents a robot. So if we have N number of robots, we need N number of processors. The ODEs in Equation (6) is then solved numerically on each processor. At each time step, the information needed is the states (the positions of each robot, in particular). So, at each time step, each robot (or processors) needs to broadcast its position to the robot/processor nearby so that that information can be used to perform the numerical calculation. In this project, three robots forming a triangular formation were studied. The parallelization approach that we implemented here is shown in Figure 4.

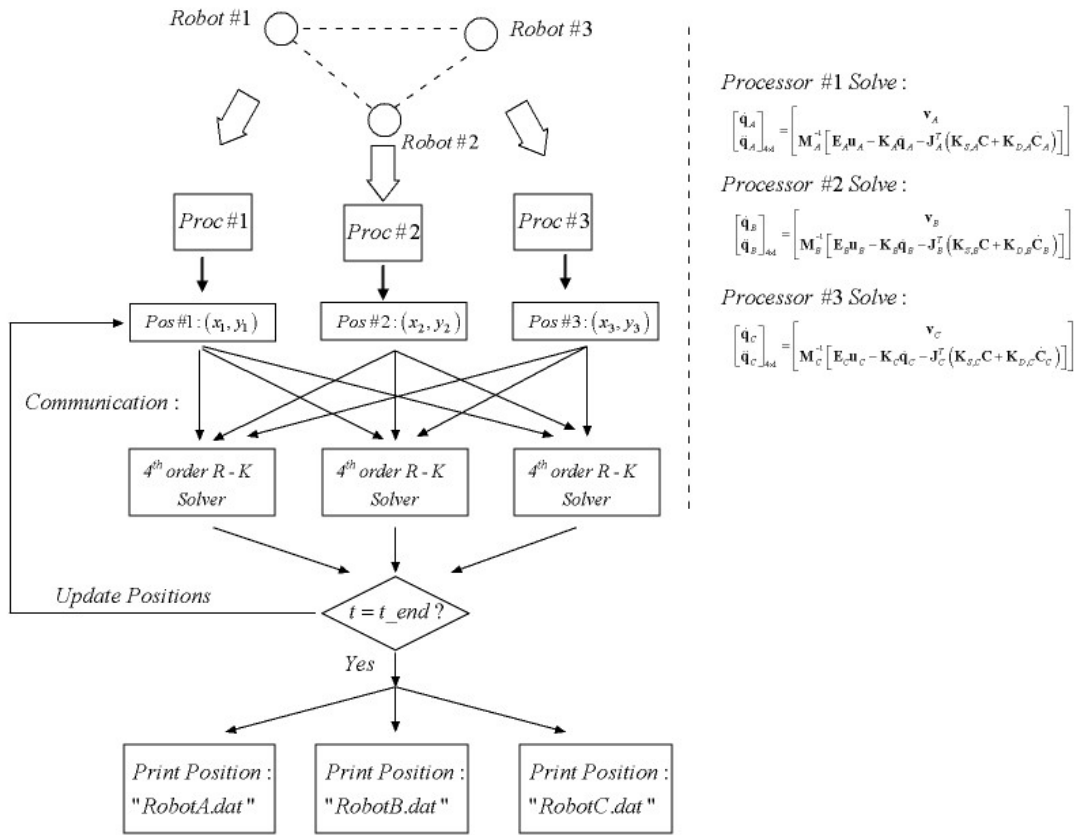


Figure 4: Parallelization scheme implemented to solve three ODE equations in a distributed manner.

Each processor has different initial positions that form a triangular formation shape. The next step, each processor send their position data to the other two processors and received the position data from the other two processors. These informations were then used in the function evaluation in the ode solver. The ode solver that we implemented here is the 4th order Runge-Kutta Method. This communication process repeated at each time step throughout the simulation. The position data for each robot were write to three different data files separately on each processor. Namely “RobotA.dat”, “RobotB.dat” and “RobotC.dat”.

ODE SOLVER:

The ODE solver package that developed for parallel processing is PETSc. We tried to implement this solver to solve the 2nd order ODE equations that we have, but the

learning curve is very steep. After two weeks of attempt, we decided to write our own ODE solver using a 4th order Runge-Kutta Method.

Suppose there are two variables, and:

$$\begin{aligned}\dot{x} &= f(t, x, y) \\ \dot{y} &= f(t, x, y)\end{aligned}\tag{8}$$

Let x_n be the value of x at time t_n , and similarly for y_n , the formula for Runge-Kutta is given by:

$$\begin{aligned}x_{n+1} &= x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + 4k_4) \\ y_{n+1} &= y_n + \frac{h}{6}(j_1 + 2j_2 + 2j_3 + 4j_4)\end{aligned}\tag{9}$$

Where:

$$\begin{aligned}k_1 &= f(t_n, x_n, y_n) & k_2 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1, y_n + \frac{h}{2}j_1\right) \\ j_1 &= f(t_n, x_n, y_n) & j_2 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1, y_n + \frac{h}{2}j_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2, y_n + \frac{h}{2}j_2\right) & k_4 &= f(t_n + h, x_n + hk_3, y_n + hj_3) \\ j_3 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2, y_n + \frac{h}{2}j_2\right) & j_4 &= f(t_n + h, x_n + hk_3, y_n + hj_3)\end{aligned}$$

These equations were used to solve the 2nd order ODE equations in this project.

IMPLEMENTATION AND RESULTS:

The passage passing between the processor is done using MPI, the code is written in C language. Two cases were studied in this project. They are in different level of difficulties: Case 1 is a workspace without an obstacle. The three robots will maintain a triangular formation and move to the target point in a quadratic potential field. In case 2, the three robots try to maintain their formation while avoiding an obstacle that lies between them and the target position. In both cases, we successfully simulate the motion planning in a distributed manner. The result (time, x and y position, x and y velocities) for each robot are saved on different files (3 files in this case). We then use MATLAB to plot the position of the robot and the corresponding contour plot of the workspace

potential to show how the simulation. These results were shown separately in the following paragraphs.

Case 1: Motion planning of three robots in formation without obstacle in a quadratic potential field.

In the first case study, three robots forming a triangular formation shape try to move to the desired target, and there are no obstacles in the work space. This situation is visualized in Figure 5 below.

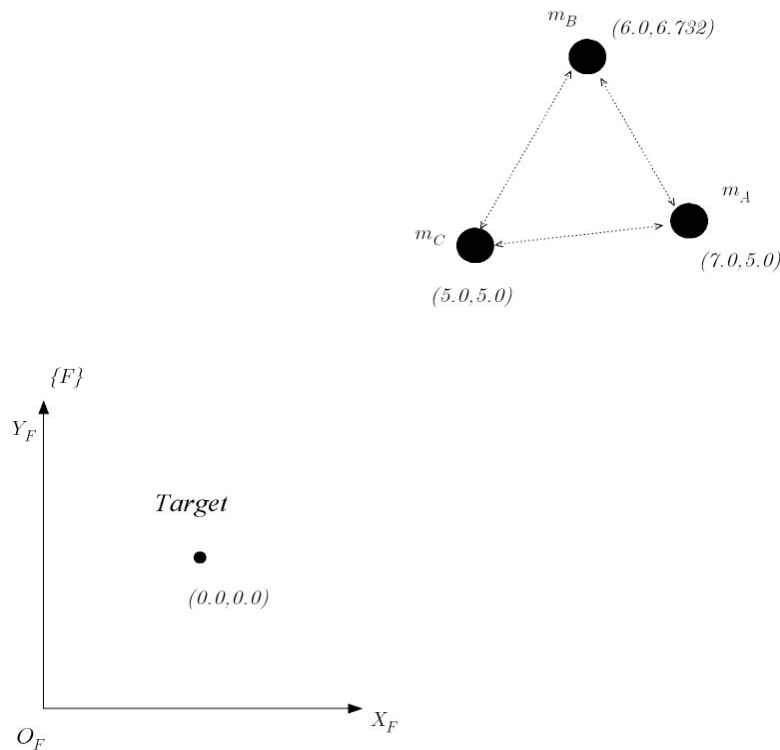


Figure 5: Motion planning of three robots forming a triangular shape formation in a workspace without obstacle.

After the simulation complete, the results are plotted using MATLAB, along with the potential field's contour plot. The result of the simulation is shown in Figure 6. The total simulation time is 8 seconds and the time step is 5×10^{-3} . The spring constant that maintain the formation is set to be 300. The positions of the three robots are represented in a triangular shape when plotted in MATLAB. The corner of the triangle is the location of the three robots at each time instant.

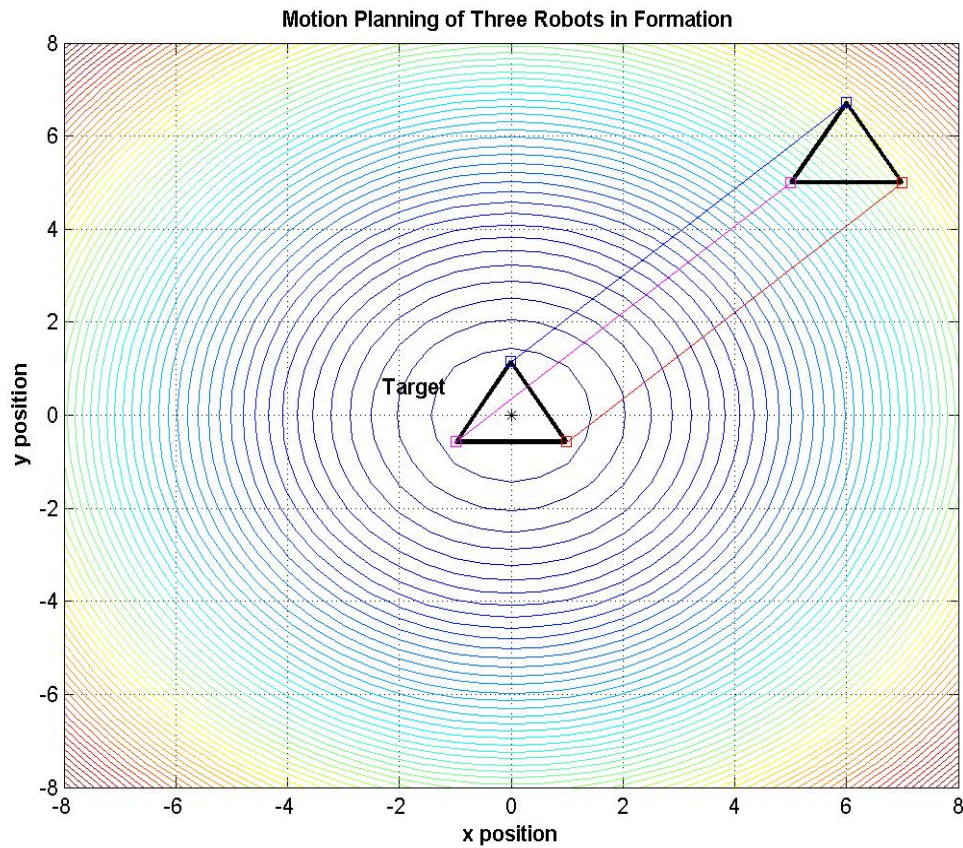


Figure 6: Motion planning of a three robot forming a triangular shape in workspace without obstacle.

From Figure 6, we can see that the robot formation successfully reached its target position, which is located at (0,0). Throughout its motion, the formation is being maintained.

Case 2: Motion planning of three robots in formation in workspace with one obstacle, potential created using navigation function.

In the second case study, the motion planning of three robots forming a triangular shape in a workspace with an obstacle is being shown in Figure 7. In this case study, the target position is located at (2.5, 2.5). The robot want to reach this position while avoiding the collision with the obstacle located at (0, 0) with radius 1.5. The potential field is generated using the navigation function that described in the previous section. The total simulation time is 10 seconds and the time-step is 5×10^{-3} . In this case, the gradient at each time step is calculated numerically. The solution of this case study is shown in Figure 8.

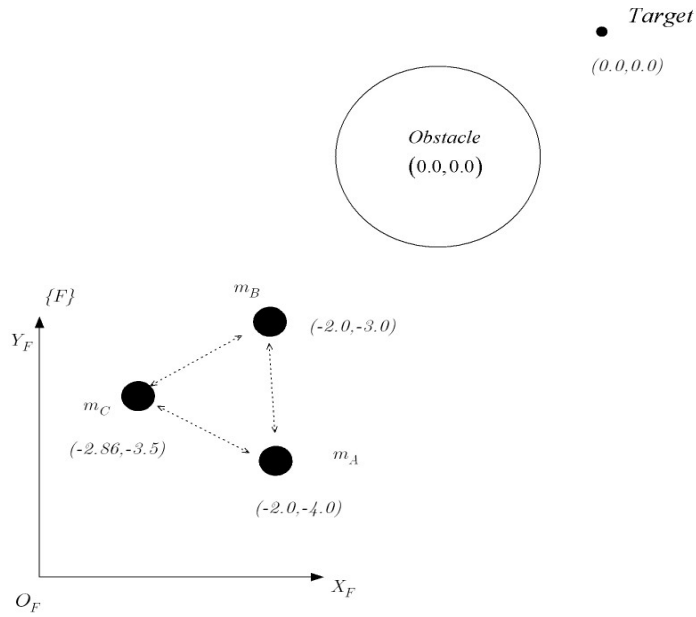


Figure 7: Motion planning of a three robot collectives in a workspace with one obstacle.

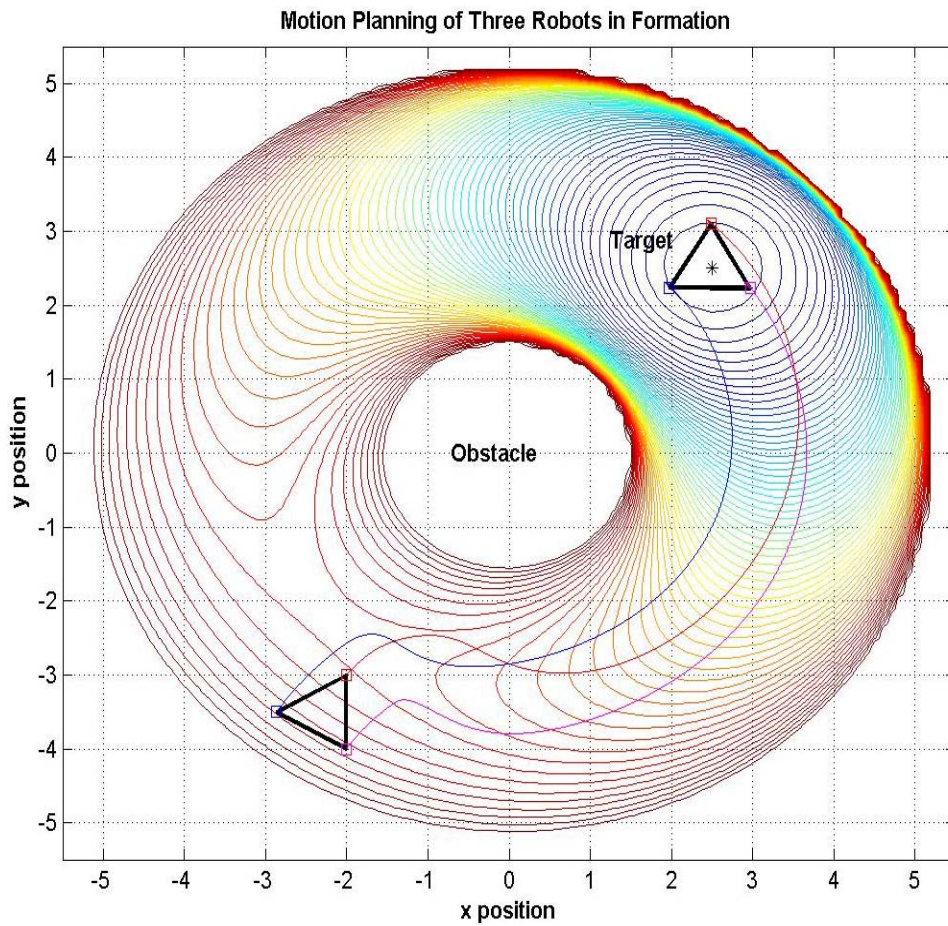


Figure 8: Result of motion planning of a three robot collectives in workspace with one obstacle.

The simulation result is shown in Figure 8. The corner of the triangular shape is the position of the robots collectives that form the triangular. In this plot, we can see that the robots collectives follow the negative gradient of the contour of the potential and reach the desired target position.

CONCLUSION AND DISCUSSION:

In this project, we successfully implemented the decentralized formulation of our motion planning scheme in a distributed manner. The penalty-based formulation approach allows the decentralized formulation of our constrained mechanical system, and thus made the distributed simulation possible. The total simulation time for both of the case study is less than 10 seconds to perform. Since the main purpose of this project is to implement the parallel processing ability of the formulation and not to study the computational speedup achieved by parallelization, the performance of such analysis is not performed.

From the result, we can see that our Runge-Kutta algorithm works well, one worth nothing is that to maintain strict formation, the spring constant value in the formulation need to set to relatively high (>1000). In this case the ODEs become a stiff equations and it is necessary to use a smaller time-step to run the simulation to ensure the stability.

In this project, we wrote the Runge-Kutta ODE solver to solve the ODEs equations in our system. It is initially intended to use the PETSc solver packages to perform this task but the author is not able to fully understand how the solver works in the given time frame for this project. Nonetheless, the Runge-Kutta solver that we wrote performed very well when we tested several known nonlinear equations with our solver and compare with the result obtained using MATLAB.

This project is completed successfully with the helpful inputs and suggestion from Dr. Patra and Dr. Jones from the initial planning stage to the actual coding stage. In summary, this project demonstrate the decentralized processing ability of the penalty-based formulation used in motion planning for a group of robots in formation using potential field approach for collision avoidance.

The codes used for these two cases studies were included for references.

```

/* -----
*   FALL 2005 HIGH PERFORMANCE COMPUTING I FINAL PROJECT
*
*   TITLE: DISTRIBUTED MOTION PLANNING OF THREE ROBOTS USING POTENTIAL
*           FIELD APPROACH
*
*   NAME: LENG-FENG LEE
*   ADVISORS: DR PATRA & DR JONES
*   DATE: 08 DECEMBER 2005
*
*   DESCRIPTION:
*   The motion planning for a group of robots with formation was formulated in
*   a decentralized manner. There are two issues involved in the formulation.
*   (1) The motion planning problem; and (2) The formation maintaining problem.
*   In our formulation, the first task was completed using Artificial Potential
*   Approach, and the second task was completed by formulating the the 3 robots
*   system as a constrained-mechanical system. In solving the resulting
*   constrained mechanical system, a penalty-formulation approach was implemented.
*   In this project, we simulate the dynamics of the three robots using this
*   approach. The dynamic equations (ODEs) were solved using a 4th order Runge-
*   Kutta method while at each iterations, the positions information on each robot/
*   processors needed to exchange such that the updated position of each robots
*   can be used in the penalty-based formulation. This project demonstrate the
*   decentralized ability of the formulation.
*
*   CASE STUDY 1: MOTION PLANNING OF THREE ROBOT COLLECTIVES IN QUADRATIC POTENTIAL
*                 FIELD WITHOUT OBSTACLE.
* ----- */

```

```

#include<stdio.h>
#include<math.h>
#include "mpi.h"

#define true 1
#define false 0

main(int argc, char** argv)
{
    double Xini,Xdotini,Yini,Ydotini;          /* Initial Conditions */
    double T;                                  /* Total Time T */
    double W1,W2,W3,W4;                        /* Variables used in Runge-Kutta Method */
    double X11,X12,X21,X22,X31,X32,X41,X42;

```

```

double Y11,Y12,Y21,Y22,Y31,Y32,Y41,Y42;

int OK=true;
FILE *OUP[1];

/* --- Function Definitions used in R-K Method ---*/
double F_Xdot(double, double, double);           /* (Time, x-position, x-
velocity) */
double F_Xdotdot(double, double, double, double [3][2], int); /* (Time, x-position, x-
velocity) */
double F_Ydot(double, double, double);           /* (Time, y-position, y-
velocity) */
double F_Ydotdot(double, double, double, double [3][2], int); /* (Time, y-position, y-
velocity) */

double absval(double);
void OUTPUT(FILE **);

/* void INPUT(int *, double *, double *, double *, double *, double *, double *, int *);*/

/* ---Initialization ----- */
double t0 = 0.0;           /* Simulation start time */
double tf = 10.0;         /* Simulation end time */
int N = 1000;             /* No. of Steps used */
double H = 0.0;           /* Size of Time-step */
int I = 0;                 /* No. of iterations */
double PositionData[2] = 0.0;
double RobotsPos[3][2] = 0.0;

/* --- Initialization used for Parallel Processing---*/
int p_rank;               /* Processor/Robot ID */
int p;                    /* Total No. of Processors/Robots */
int source = 0;           /* Default value for message source */
int tag = 0;              /* Tag ID for message passing */
MPI_Status status;       /* Used for message passing */

/* Start up MPI */
MPI_Init(&argc, &argv);

/* Get the processors/robot rank */
MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);

```

```

/* Get the total number of processor/robots are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

/* --- Initialize the location of each robots ---*/
Xdotini = 0.0; /* All robots start from rest initially */
Ydotini = 0.0;
if (p_rank == 0) /* Initial positions of Robot 0 */
{
    Xini = 5.0;
    Yini = 5.0;
}
if (p_rank == 1) /* Initial positions of Robot 1 */
{
    Xini = 7.0;
    Yini = 5.0;
}
if (p_rank == 2) /* Initial positions of Robot 2 */
{
    Xini = 6.0;
    Yini = 6.73205;
}

/*INPUT(&OK, &A, &B, &Xini, &Xdotini, &Yini, &Ydotini, &N)*/;

if (OK) {
    OUTPUT(OUP);
    /* STEP 1 */
    H = (tf - t0) / N;
    T = t0;
    /* STEP 2 */
    W1 =Xini; /* Initial X value */
    W2 =Xdotini; /* Initial Xdot value */
    W3 =Yini; /* Initial Y value */
    W4 =Xdotini; /* Initial Ydot value */

    /* STEP 3 */
    fprintf(*OUP, "%5.3f %11.8f %11.8f %11.8f %11.8f\n", T, W1, W2, W3, W4);

    /* STEP 4 */
    for (I=1; I<=N; I++) {

```

```

/* --- Communication begins ----*/
PositionData[0]=W1;          /* Pack the position data in a vec */
PositionData[1]=W3;
for(int i=0; i <p; i++)
{
    if (i != p_rank)
    {
        MPI_Send(PositionData, 2, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
    }
}
for (int i=0; i<p; i++)
{
    if (i != p_rank)
    {
        MPI_Recv(RobotsPos[i], 2, MPI_DOUBLE, i, tag, MPI_COMM_WORLD, &status);
    }
}

/* --- Communication Ends -----*/

/* STEP 5 */
X11 = H * F_Xdot(T, W1, W2);
X12 = H * F_Xdotdot(T, W1, W2, RobotsPos[][2], p_rank);
Y11 = H * F_Ydot(T, W3, W4);
Y12 = H * F_Ydotdot(T, W3, W4, RobotsPos[][2], p_rank);

/* STEP 6 */
X21 = H * F_Xdot(T + H / 2.0, W1 + X11 / 2.0, W2 + X12 / 2.0);
X22 = H * F_Xdotdot(T + H / 2.0, W1 + X11 / 2.0, W2 + X12 / 2.0, RobotsPos[][2], p_rank);
Y21 = H * F_Ydot(T + H / 2.0, W3 + Y11 / 2.0, W4 + Y12 / 2.0);
Y22 = H * F_Ydotdot(T + H / 2.0, W3 + Y11 / 2.0, W4 + Y12 / 2.0, RobotsPos[][2], p_rank);

/* STEP 7 */
X31 = H * F_Xdot(T + H / 2.0, W1 + X21 / 2.0, W2 + X22 / 2.0);
X32 = H * F_Xdotdot(T + H / 2.0, W1 + X21 / 2.0, W2 + X22 / 2.0, RobotsPos[][2], p_rank);
Y31 = H * F_Ydot(T + H / 2.0, W3 + Y21 / 2.0, W4 + Y22 / 2.0);
Y32 = H * F_Ydotdot(T + H / 2.0, W3 + Y21 / 2.0, W4 + Y22 / 2.0, RobotsPos[][2], p_rank);

/* STEP 8 */
X41 = H * F_Xdot(T + H, W1 + X31, W2 + X32);
X42 = H * F_Xdotdot(T + H, W1 + X31, W2 + X32, RobotsPos[][2], p_rank);
Y41 = H * F_Ydot(T + H, W3 + Y31, W4 + Y32);

```

```

Y42 = H * F_Ydotdot(T + H, W3 + Y31, W4 + Y32, RobotsPos[][2], p_rank);

/* STEP 9 */
W1 = W1 + (X11 + 2.0 * X21 + 2.0 * X31 + X41) / 6.0;
W2 = W2 + (X12 + 2.0 * X22 + 2.0 * X32 + X42) / 6.0;
W3 = W3 + (Y11 + 2.0 * Y21 + 2.0 * Y31 + Y41) / 6.0;
W4 = W4 + (Y12 + 2.0 * Y22 + 2.0 * Y32 + Y42) / 6.0;

/* STEP 10 */
T = A + I * H;

/* STEP 11 */
fprintf(*OUP, "%5.3f %11.8f %11.8f %11.8f %11.8f \n", T, W1, W2, W3, W4);

}

/* STEP 12 */
fclose(*OUP);
}
return 0;
}

/* ----- FUNCTION EVALUATIONS ----- */
double F_Xdot(double T, double X1, double X2)
{
    double f;

    f = X2;
    return f;
}

double F_Xdotdot(double T, double X1, double X2, double RobPos[][2], int prank)
{
    double f;
    double x_tar=0;
    double k = 3.0;
    double M;
    double Ks = 100; /*Spring Constant*/
    double C1, C2, C3;

    C1=(RobPos[0][0]-RobPos[1][0])*(RobPos[0][0]-RobPos[1][0]) + (RobPos[0][1]-RobPos[1][1])*(RobPos[0]
[1]-RobPos[1][1])-R*R;
    C2=(RobPos[1][0]-RobPos[2][0])*(RobPos[1][0]-RobPos[2][0]) + (RobPos[1][1]-RobPos[2][1])*(RobPos[1]

```

```

[1]-RobPos[2][1])-R*R;
    C3=(RobPos[2][0]-RobPos[0][0])*(RobPos[2][0]-RobPos[0][0]) + (RobPos[2][1]-RobPos[0][1])*(RobPos[2]
[1]-RobPos[0][1])-R*R;

    if (prank == 0)
        M = (RobPos[0][0]-RobPos[1][0])*Ks*C1 - (RobPos[2][0]-RobPos[0][0])*Ks*C3;
    if (prank == 1)
        M = (RobPos[1][0]-RobPos[2][0])*Ks*C1 - (RobPos[0][0]-RobPos[1][0])*Ks*C3;
    if (prank == 2)
        M = (RobPos[2][0]-RobPos[0][0])*Ks*C1 - (RobPos[1][0]-RobPos[2][0])*Ks*C3;

    f = (x_tar-X1)-k*X2-M;
    return f;
}

double F_Ydot(double T, double Y1, double Y2)
{
    double f;

    f = Y2;
    return f;
}

double F_Ydotdot(double T, double Y1, double Y2, double RobPos[][2], int prank)
{
    double f;
    double y_tar=0;
    double k = 3.0;
    double C1,C2,C3;

    C1=(RobPos[0][0]-RobPos[1][0])*(RobPos[0][0]-RobPos[1][0]) + (RobPos[0][1]-RobPos[1][1])*(RobPos[0]
[1]-RobPos[1][1])-R*R;
    C2=(RobPos[1][0]-RobPos[2][0])*(RobPos[1][0]-RobPos[2][0]) + (RobPos[1][1]-RobPos[2][1])*(RobPos[1]
[1]-RobPos[2][1])-R*R;
    C3=(RobPos[2][0]-RobPos[0][0])*(RobPos[2][0]-RobPos[0][0]) + (RobPos[2][1]-RobPos[0][1])*(RobPos[2]
[1]-RobPos[0][1])-R*R;

    if (prank == 0)
        M = (RobPos[0][1]-RobPos[1][1])*Ks*C1 - (RobPos[2][1]-RobPos[0][1])*Ks*C3;
    if (prank == 1)
        M = (RobPos[1][1]-RobPos[2][1])*Ks*C1 - (RobPos[0][1]-RobPos[1][1])*Ks*C3;
    if (prank == 2)
        M = (RobPos[2][1]-RobPos[0][1])*Ks*C1 - (RobPos[1][1]-RobPos[2][1])*Ks*C3;
}

```

```

    f = (y_tar-Y1)-k*Y2;
    return f;
}

/* ----- */

/*void INPUT(int *OK, double *A, double *B, double *Xini, double *Xdotini, double *Yini, double
*Ydotini, int *N)
{
    double X;
    char AA;

    printf("This is the Runge-Kutta Method for Systems with m = 2.\n");
    *OK = false;
    printf("Have the functions F1 and F2 been defined? ");
    printf("Answer Y or N.\n");
    scanf("%c",&AA);
    if ((AA == 'Y') || (AA == 'y')) {
        *OK = false;
        while (!(*OK)) {
            printf("Input start time and end time separated by blank\n");
            scanf("%lf %lf", A, B);
            if (*A >= *B)
                printf("Left endpoint must be less than right endpoint\n");
            else *OK = true;
        }
        printf("Input the two initial conditions, separated by blank.\n");
        scanf("%lf %lf %lf %lf", Xini, Xdotini, Yini, Ydotini);
        *OK = false;
        while(!(*OK)) {
            printf("Input a positive integer for the number of subintervals\n");
            scanf("%d", N);
            if (*N <= 0) printf("Number must be a positive integer\n");
            else *OK = true;
        }
    }
    else {
        printf("The program will end so that the functions can be created.\n");
        *OK = false;
    }
}*/

```

```

void OUTPUT(FILE **OUP)
{
    char NAME[30];
    int FLAG;

    printf("Choice of output method:\n");
    printf("1. Output to screen\n");
    printf("2. Output to text File\n");
    printf("Please enter 1 or 2\n");
    scanf("%d", &FLAG);
    if (FLAG == 2) {
        printf("Input the file name in the form - drive:name.ext\n");
        printf("For example A:OUTPUT.DTA\n");
        scanf("%s", NAME);
        *OUP = fopen(NAME, "w");
    }
    else *OUP = stdout;
    fprintf(*OUP, "RUNGE-KUTTA METHOD FOR SYSTEMS WITH m = 2.\n");
    fprintf(*OUP, "      T           W1           W2           W3           W4\n\n");
}

```

```

/* -----
*   FALL 2005 HIGH PERFORMANCE COMPUTING I FINAL PROJECT
*
*   TITLE: DISTRIBUTED MOTION PLANNING OF THREE ROBOTS USING POTENTIAL
*           FIELD APPROACH
*
*   NAME: LENG-FENG LEE
*   ADVISORS: DR PATRA & DR JONES
*   DATE: 08 DECEMBER 2005
*
*   DESCRIPTION:
*   The motion planning for a group of robots with formation was formulated in
*   a decentralized manner. There are two issues involved in the formulation.
*   (1) The motion planning problem; and (2) The formation maintaining problem.
*   In our formulation, the first task was completed using Artificial Potential
*   Approach, and the second task was completed by formulating the the 3 robots
*   system as a constrained-mechanical system. In solving the resulting
*   constrained mechanical system, a penalty-formulation approach was implemented.
*   In this project, we simulate the dynamics of the three robots using this
*   approach. The dynamic equations (ODEs) were solved using a 4th order Runge-
*   Kutta method while at each iterations, the positions information on each robot/
*   processors needed to exchange such that the updated position of each robots
*   can be used in the penalty-based formulation. This project demonstrate the
*   decentralized ability of the formulation.
*
*   CASE STUDY 2: MOTION PLANNING OF THREE ROBOT COLLECTIVES IN NAVIGATION FUNCTION
*                 POTENTIAL FIELD WITH ONE OBSTACLE.
* ----- */

```

```

#include<stdio.h>
#include<math.h>
#include "mpi.h"

```

```

#define true 1
#define false 0
#define PI 3.14159263589

```

```

main(int argc, char** argv)

```

```

{
    double Xini,Xdotini,Yini,Ydotini;          /* Initial Conditions */
    double T;                                  /* Total Time T */
    double W1,W2,W3,W4;                        /* Variables used in Runge-Kutta Method */
    double X11,X12,X21,X22,X31,X32,X41,X42;

```

```

double Y11,Y12,Y21,Y22,Y31,Y32,Y41,Y42;

int OK=true;
FILE *OUP[1];

/* --- ODEs Function Definitions used in R-K Method ---*/
double F_Xdot(double, double, double); /* (Time, x-position, x-
velocity) */
double F_Xdotdot(double, double, double, double [3][2], int); /* (Time, x-position, x-
velocity) */
double F_Ydot(double, double, double); /* (Time, y-position, y-
velocity) */
double F_Ydotdot(double, double, double, double [3][2], int); /* (Time, y-position, y-
velocity) */

/*double QuadGradient(double [2], double [3][2], int);*/

void OUTPUT(FILE **, int);

/* void INPUT(int *, double *, double *, double *, double *, double *, double *, int *);*/

/* ---Initialization ----- */
double t0 = 0.0; /* Simulation start time */
double tf = 8.0; /* Simulation end time */
int N = 16000; /* No. of Steps used */
double H = 0.0; /* Size of Time-step */
int I = 0; /* No. of iterations */
double PositionData[2] = {0.0, 0.0};
double RobotsPos[3][2] = {{0.0,0.0},{0.0,0.0},{0.0,0.0}};
int i,j;

/* --- Initialization used for Parallel Processing---*/
int p_rank; /* Processor/Robot ID */
int p; /* Total No. of Processors/Robots */
int source = 0; /* Default value for message source */
int tag = 0; /* Tag ID for message passing */
MPI_Status status; /* Used for message passing */

/* Start up MPI */
MPI_Init(&argc, &argv);

/* Get the processors/robot rank */

```

```

MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);

/* Get the total number of processor/robots are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

/* --- Initialize the location of each robots ---*/
Xdotini = 0.0; /* All robots start from rest initially */
Ydotini = 0.0;
if (p_rank == 0) /* Initial positions of Robot 0 */
{
    Xini = -2.0;
    Yini = -4.0;
}
if (p_rank == 1) /* Initial positions of Robot 1 */
{
    Xini = -2.0;
    Yini = -3.0;
}
if (p_rank == 2) /* Initial positions of Robot 2 */
{
    Xini = -2.866025;
    Yini = -3.5;
}
/* --- End Initialized Robot position ----- */

if (OK) {
    OUTPUT(OUP, p_rank);
    /* STEP 1 */
    H = (tf - t0) / N;
    T = t0;
    /* STEP 2 */
    W1 =Xini; /* Initial X value */
    W2 =Xdotini; /* Initial Xdot value */
    W3 =Yini; /* Initial Y value */
    W4 =Ydotini; /* Initial Ydot value */

    /* STEP 3 */
    printf("%5.3f %11.8f %11.8f %11.8f %11.8f\n", T, W1, W2, W3, W4);
    fprintf(*OUP, "%5.3f %11.8f %11.8f %11.8f %11.8f\n", T, W1, W2, W3, W4);

    /* STEP 4 */
    for (I=1; I<=N; I++) {

```

```

/* --- Communication begins ----*/
PositionData[0]=W1;          /* Pack the position data in a vec */
PositionData[1]=W3;
RobotsPos[p_rank][0]=W1;
RobotsPos[p_rank][1]=W3;
for(i=0; i <p; i++)
{
    if (i != p_rank)
    {
        MPI_Send(PositionData, 2, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);

    }
}
/*printf("Hello, we have send out!!\n\n");*/
for ( j=0; j<p; j++)
{
    if (j != p_rank)
    {
        MPI_Recv(RobotsPos[j], 2, MPI_DOUBLE, j, tag, MPI_COMM_WORLD, &status);
    }
}
/*printf("Hello, we have received!!\n\n");*/

/* --- Communication Ends -----*/

/* Check point #1*/
/*if (p_rank==0)
{
    for (i=0; i<p; i++)
    {
        for (j=0;j<2;j++)
            printf("%4.6f  ",RobotsPos[i][j]);
        printf("\n");
    }
}*/
/* End Check Point #1*/
/* STEP 5 */
X11 = H * F_Xdot(T, W1, W2);
X12 = H * F_Xdotdot(T, W1, W2, RobotsPos, p_rank);
Y11 = H * F_Ydot(T, W3, W4);
Y12 = H * F_Ydotdot(T, W3, W4, RobotsPos, p_rank);

```

```

/* STEP 6 */
X21 = H * F_Xdot(T + H / 2.0, W1 + X11 / 2.0, W2 + X12 / 2.0);
X22 = H * F_Xdotdot(T + H / 2.0, W1 + X11 / 2.0, W2 + X12 / 2.0, RobotsPos, p_rank);
Y21 = H * F_Ydot(T + H / 2.0, W3 + Y11 / 2.0, W4 + Y12 / 2.0);
Y22 = H * F_Ydotdot(T + H / 2.0, W3 + Y11 / 2.0, W4 + Y12 / 2.0, RobotsPos, p_rank);

/* STEP 7 */
X31 = H * F_Xdot(T + H / 2.0, W1 + X21 / 2.0, W2 + X22 / 2.0);
X32 = H * F_Xdotdot(T + H / 2.0, W1 + X21 / 2.0, W2 + X22 / 2.0, RobotsPos, p_rank);
Y31 = H * F_Ydot(T + H / 2.0, W3 + Y21 / 2.0, W4 + Y22 / 2.0);
Y32 = H * F_Ydotdot(T + H / 2.0, W3 + Y21 / 2.0, W4 + Y22 / 2.0, RobotsPos, p_rank);

/* STEP 8 */
X41 = H * F_Xdot(T + H, W1 + X31, W2 + X32);
X42 = H * F_Xdotdot(T + H, W1 + X31, W2 + X32, RobotsPos, p_rank);
Y41 = H * F_Ydot(T + H, W3 + Y31, W4 + Y32);
Y42 = H * F_Ydotdot(T + H, W3 + Y31, W4 + Y32, RobotsPos, p_rank);

/* STEP 9 */
W1 = W1 + (X11 + 2.0 * X21 + 2.0 * X31 + X41) / 6.0;
W2 = W2 + (X12 + 2.0 * X22 + 2.0 * X32 + X42) / 6.0;
W3 = W3 + (Y11 + 2.0 * Y21 + 2.0 * Y31 + Y41) / 6.0;
W4 = W4 + (Y12 + 2.0 * Y22 + 2.0 * Y32 + Y42) / 6.0;

/* STEP 10 */
T = t0 + I * H;

/* STEP 11 */
/*if (p_rank ==0)
    printf("%5.3f %11.8f %11.8f %11.8f %11.8f \n", T, W1, W2, W3, W4);*/
fprintf(*OUP, "%5.3f %11.8f %11.8f %11.8f %11.8f \n", T, W1, W2, W3, W4);

} /* End for-loop */

/* STEP 12 */
fclose(*OUP);
}

MPI_Finalize();

return 0;
}

```

```

/* ----- FUNCTION EVALUATIONS ----- */
double F_Xdot(double T, double X1, double X2)
{
    double f;

    f = X2;
    return f;
}

double F_Xdotdot(double T, double X1, double X2, double RobPos[][2], int prank)
{
    double f;
    double xtar = 2.5;          /* Target X-position */
    double ytar = 2.5;          /* Target Y-Position */
    double kp = 3.0;
    double M;
    double Ks = 3000;          /*Spring Constant*/
    double C1, C2, C3;
    double R=1;

    C1=(RobPos[0][0]-RobPos[1][0])*(RobPos[0][0]-RobPos[1][0]) + (RobPos[0][1]-RobPos[1][1])*(RobPos[0][1]-RobPos[1][1])-R*R;
    C2=(RobPos[1][0]-RobPos[2][0])*(RobPos[1][0]-RobPos[2][0]) + (RobPos[1][1]-RobPos[2][1])*(RobPos[1][1]-RobPos[2][1])-R*R;
    C3=(RobPos[2][0]-RobPos[0][0])*(RobPos[2][0]-RobPos[0][0]) + (RobPos[2][1]-RobPos[0][1])*(RobPos[2][1]-RobPos[0][1])-R*R;

    if (prank == 0)
        M = (RobPos[0][0]-RobPos[1][0])*Ks*C1 - (RobPos[2][0]-RobPos[0][0])*Ks*C3;
    if (prank == 1)
        M = (RobPos[1][0]-RobPos[2][0])*Ks*C2 - (RobPos[0][0]-RobPos[1][0])*Ks*C1;
    if (prank == 2)
        M = (RobPos[2][0]-RobPos[0][0])*Ks*C3 - (RobPos[1][0]-RobPos[2][0])*Ks*C2;

    /* --- Gradient finding using Navigation Function ---- */
    int N = 1;                  /* No. of Obstatcle */
    double xobs = 0.0;          /* x-location of the obstacle */
    double yobs = 0.0;          /* y-location of the obstacle */
    double robs = 1.5;          /* obstacle radius */
    double xbon = 0.0;          /* bounding sphere x-location */
    double ybon = 0.0;          /* bounding sphere y-location */
    double rbon = 5.2;          /* bounding sphere radius */
    double k = 1.6;

```

```

double xrob = RobPos[prank][0]; /* Current robot x position */
double yrob = RobPos[prank][1]; /* Current robot y position */

double GradientTest = 0.0;

double Beta = ((xrob-xobs)*(xrob-xobs) + (yrob-yobs)*(yrob-yobs))-robs*robs;
double BetaB= -((xrob-xbon)*(xrob-xbon) + (yrob-ybon)*(yrob-ybon))+rbon*rbon;
double BETA = Beta*BetaB;
double Norm = pow( (xrob-xtar)*(xrob-xtar) + (yrob-ytar)*(yrob-ytar), 0.5 );
double GradientNegative = 0;
double Ux, Uy, Gradient, NaFunc, Numerator, Denominator, temp;

if (BETA > 0)
{
    double Numerator = Norm*Norm;
    double temp = ( pow(Norm,2*k) + BETA );
    double Denominator = pow(temp, 1/k);
    NaFunc = Numerator/Denominator;
    double ang, delta, xrob_new, yrob_new, Betanew, BetaBnew, BETAnew, Norm_new, Numerator_new;
    double Denominator_new, NaFunc_new;

    for (ang=0; ang<2*PI; ang=ang+(PI/180))
    {
        delta = 0.1;
        xrob_new = xrob+delta*cos(ang);
        yrob_new = yrob+delta*sin(ang);

        Betanew = ((xrob_new-xobs)*(xrob_new-xobs) + (yrob_new-yobs)*(yrob_new-yobs))-robs*robs;
        BetaBnew= -((xrob_new-xbon)*(xrob_new-xbon) + (yrob_new-ybon)*(yrob_new-ybon))+rbon*rbon;
        BETAnew = Betanew*BetaBnew;
        Norm_new = pow( (xrob_new-xtar)*(xrob_new-xtar) + (yrob_new-ytar)*(yrob_new-ytar), 0.5 );
        Numerator_new = Norm_new*Norm_new;
        temp = ( pow(Norm_new,2*k) + BETAnew );
        Denominator_new = pow(temp, 1/k);
        NaFunc_new = Numerator_new/Denominator_new;
        Gradient = (NaFunc_new - NaFunc)/delta;

        if ( Gradient < GradientTest )
        {
            Ux = cos(ang);
            Uy = sin(ang);
            GradientTest = Gradient;
        }
    }
}

```

```

        GradientNegative = 1;
    }
    if (GradientNegative == 0)
    {
        Ux = 0.0;
        Uy = 0.0;
    }
}
}
/* --- End Gradient Finding ----- */
/*printf("Gradient are %f   %f from processor %d\n",Ux,Uy, prank);*/
f =5*Ux-kp*X2-M;
/*f = 6*(xtar-X1)-kp*X2-M;*/
return f;
}

double F_Ydot(double T, double Y1, double Y2)
{
    double f;

    f = Y2;
    return f;
}

double F_Ydotdot(double T, double Y1, double Y2, double RobPos[][2], int prank)
{
    double f;
    double xtar = 2.5;          /* Target X-position */
    double ytar = 2.5;          /* Target Y-Position */
    double kp = 3.0;            /* Constant proportional force term */
    double C1,C2,C3;            /* Euclidean distant between each robot */
    double M;                    /* Temp variable */
    double Ks=3000;              /* Spring-Constant of the constraint */
    double R=1;                  /* Desired Distant between robot */

    C1=(RobPos[0][0]-RobPos[1][0])*(RobPos[0][0]-RobPos[1][0]) + (RobPos[0][1]-RobPos[1][1])*(RobPos[0][1]-RobPos[1][1])-R*R;
    C2=(RobPos[1][0]-RobPos[2][0])*(RobPos[1][0]-RobPos[2][0]) + (RobPos[1][1]-RobPos[2][1])*(RobPos[1][1]-RobPos[2][1])-R*R;
    C3=(RobPos[2][0]-RobPos[0][0])*(RobPos[2][0]-RobPos[0][0]) + (RobPos[2][1]-RobPos[0][1])*(RobPos[2][1]-RobPos[0][1])-R*R;

    if (prank == 0)

```

```

    M = (RobPos[0][1]-RobPos[1][1])*Ks*C1 - (RobPos[2][1]-RobPos[0][1])*Ks*C3;
if (prank == 1)
    M = (RobPos[1][1]-RobPos[2][1])*Ks*C2 - (RobPos[0][1]-RobPos[1][1])*Ks*C1;
if (prank == 2)
    M = (RobPos[2][1]-RobPos[0][1])*Ks*C3 - (RobPos[1][1]-RobPos[2][1])*Ks*C2;

/* --- Gradient finding using Navigation Function ---- */
int N = 1;          /* No. of Obstatcle */
double xobs = 0.0;  /* x-location of the obstacle */
double yobs = 0.0;  /* y-location of the obstacle */
double robs = 1.5;  /* obstacle radius */
double xbon = 0.0;  /* bounding sphere x-location */
double ybon = 0.0;  /* bounding sphere y-location */
double rbon = 5.2;  /* bounding sphere radius */
double k = 1.6;

double xrob = RobPos[prank][0]; /* Current robot x position */
double yrob = RobPos[prank][1]; /* Current robot y position */

double Umin = 10.0;          /* Minimum potential */
double GradientTest = 0.0;

double Beta = ((xrob-xobs)*(xrob-xobs) + (yrob-yobs)*(yrob-yobs))-robs*robs;
double BetaB= -((xrob-xbon)*(xrob-xbon) + (yrob-ybon)*(yrob-ybon))+rbon*rbon;
double BETA = Beta*BetaB;
double Norm = pow( (xrob-xtar)*(xrob-xtar) + (yrob-ytar)*(yrob-ytar), 0.5 );
double GradientNegative = 0;
double Ux, Uy, Gradient, NaFunc, Numerator, Denominator, temp;

if (BETA > 0)
{
    Numerator = Norm*Norm;
    temp = ( pow(Norm,2*k) + BETA );
    Denominator = pow(temp, (1/k));
    NaFunc = Numerator/Denominator;
    /*printf("NavFunc is %f from processor %d",NaFunc,prank);*/

    double ang, delta, xrob_new, yrob_new, Betanew, BetaBnew, BETAnew, Norm_new, Numerator_new;
    double Denominator_new, NaFunc_new;

    for (ang=0; ang<2.1*PI; ang=ang+(PI/180))
    {
        delta = 0.1;

```

```

xrob_new = xrob + delta*cos(ang);
yrob_new = yrob + delta*sin(ang);
/*printf("xnew is %f,ynew is %f\n",xrob_new, yrob_new);*/

Betanew = ((xrob_new-xobs)*(xrob_new-xobs) + (yrob_new-yobs)*(yrob_new-yobs))-robs*robs;
/*printf("Betanew is %f", Betanew); */
BetaBnew= -((xrob_new-xbon)*(xrob_new-xbon) + (yrob_new-ybon)*(yrob_new-ybon))+rbon*rbon;
/*printf("BetaBnew is %f", BetaBnew);*/
BETAnew = Betanew*BetaBnew;
/*printf("BETAnew is %f", BETAnew);*/
Norm_new = pow( (xrob_new-xtar)*(xrob_new-xtar) + (yrob_new-ytar)*(yrob_new-ytar), 0.5 );
/*printf(" Normnew is %f" , Norm_new);*/
Numerator_new = Norm_new*Norm_new;
/*printf(" Numeratornew is %f" , Numerator_new);*/
temp = ( pow(Norm_new,2*k) + BETAnew );
Denominator_new = pow(temp, 1/k);
/*printf("Denonew is %f" , Denominator_new);*/
NaFunc_new = Numerator_new/Denominator_new;
/*printf("\nNew NavFunc is %f from %d\n",NaFunc_new,prank);*/

Gradient = (NaFunc_new - NaFunc)/delta;

if ( Gradient < GradientTest )
{
    Ux = cos(ang);
    Uy = sin(ang);
    GradientTest = Gradient;
    GradientNegative = 1;
}
if (GradientNegative == 0)
{
    Ux = 0.0;
    Uy = 0.0;
}
}
}
/* --- End Gradient Finding ----- */
/*printf("Gradient are %f %f from processor %d\n",Ux,Uy, prank);*/
/*printf("M is %f from processor %d\n", M, prank);*/
f = 5*Uy-kp*Y2-M;
/*f = 6*(ytar-Y1)-kp*Y2-M;*/
return f;
}

```

```
/* ----- */
/*double QuadGradient(double delU[2], double Pos[][2], int prank)
{
    double xtar = 0.0;
    double ytar = 0.0;

    delU[0]=(xtar-Pos[prank][0]);
    delU[1]=(ytar-Pos[prank][1]);

    return 0;
}*/

void OUTPUT(FILE **OUP, int prank)
{
    /* Save the data into three files */
    if (prank == 0)
        *OUP = fopen("RobotA.dat", "w");
    if (prank == 1)
        *OUP = fopen("RobotB.dat", "w");
    if (prank == 2)
        *OUP = fopen("RobotC.dat", "w");
}
```