AE483 Lab Manual: Appendix G (How to Add an Obstacle and a Planner)

You have previously used MOCAP to get the position and orientation of the drone. You can also use MOCAP to get the position and orientation of other things, for example "obstacles," as long as they, too, are covered in markers. In particular, this appendix describes how to modify off-board code in order to get the position of a single obstacle.

You have previously implemented a controller that achieves hover at the position (and perhaps yaw angle) defined by `QuadDesired`. This appendix also describes how to modify off-board code in order to give you a place to implement a "planner," the purpose of which is to change `QuadDesired` so that the drone reaches some goal position (and perhaps yaw angle) defined by `QuadGoal` while avoiding the obstacle defined by `ObstCurrent`.

It is really wonderful that you don't have to make _any_ changes to your on-board code in order to enable these two things!

At the end of this appendix, you will find a list of a few relevant differences between MATLAB and C that may help you when implementing a planner.

How to add an obstacle

In `main.c` of the ground station code, change

```
#define QUAD_ID                                 1   // the id you must assign to the drone when initializing MOCAP
```

to

```
#define QUAD_ID                                 1   // the id you must assign to the drone when initializing MOCAP
#define OBST_ID             2   // the id you must assign to the obstacle when initializing MOCAP
```

Naturally, when you initialize MOCAP, you must now define two rigid objects instead of one---the first object (with ID equal to `QUAD_ID`) is the drone and the second object (with ID equal to `OBST_ID` is the obstacle).

Next, change

```
// Desired position and yaw angle of quadrotor
PosYaw_f QuadDesired;
// Current position and orientation (yaw, pitch, roll) of quadrotor
State6_f QuadCurrent;
```

to

```
// Desired position and yaw angle of quadrotor
PosYaw_f QuadDesired;
// Position and orientation (yaw, pitch, roll) of quadrotor
State6_f QuadCurrent;
// Position and orientation (yaw, pitch, roll) of obstacle
State6_f ObstCurrent;
```

This gives your code a place to store the position and orientation of the obstacle, just like your code already stores the position and orientation of the drone.

Next, change

```
//update Quadrotor state
if (QUAD_ID > 0 && ID == QUAD_ID)
{
```

```
    Planner_MakeState6f_PosQuat(&QuadCurrent, &Position, &Quaternion);
    data_updated = 1;
}
```

to

```
if (QUAD_ID > 0 && ID == QUAD_ID) {
    // update quadrotor state
    Planner_MakeState6f_PosQuat(&QuadCurrent, &Position, &Quaternion);
    data_updated = 1;
} else if (OBST_ID > 0 && ID == OBST_ID) {
    // update obstacle state
    Planner_MakeState6f_PosQuat(&ObstCurrent, &Position, &Quaternion);
    data_updated = 1;
}
```

This checks if the rigid body found by MOCAP has an ID that corresponds to either your drone or your obstacle, and---if so---stores its position and orientation either in `QuadCurrent` or `ObstCurrent`, respectively.

Finally, change the `fprintf` lines so you log the position of the obstacle (`ObstCurrent.Pos.x` and so forth). No need to log the orientation if you are assuming that the obstacle is spherical---just position is fine.

Don't forget to initialize the obstacle as a rigid body with ID 2 before running this code.

How to add a planner

In `main.c` of the ground station code, change

```
// Desired position and yaw angle of quadrotor
PosYaw_f QuadDesired;
```

to

```
// Goal position and yaw angle of quadrotor
PosYaw_f QuadGoal;
// Desired position and yaw angle of quadrotor
PosYaw_f QuadDesired;
```

This defines the goal position (and yaw angle). Remember---the current position `QuadCurrent` is where the drone is now, the desired position `QuadDesired` is where you _want_ the drone to be now (using the hover controller), and the goal position `QuadGoal` is where you want the drone eventually to end up without colliding with obstacles (using the planner).

Replace the lines that set the desired position with either something like this

```
if (1) {
  // choose the goal position
  QuadGoal.yaw = 0.0;
  if (timer <= 5.0) {          // stay on ground for 5 seconds
    QuadGoal.Pos.x = 0.0;
    QuadGoal.Pos.y = 0.0;
    QuadGoal.Pos.z = 0.0;
  } else if (timer <= 45.0) {   // hover for 45 - 5 = 40 seconds
    QuadGoal.Pos.x = 0.0;
    QuadGoal.Pos.y = 0.0;
```

```
  QuadGoal.Pos.z = -1.0;
 } else {                  // back on ground forever after
  QuadGoal.Pos.x = 0.0;
  QuadGoal.Pos.y = 0.0;
  QuadGoal.Pos.z = 0.0;
 }

 // run the planner to update the desired position
 planner(QuadCurrent, ObstCurrent, &QuadDesired,  QuadGoal);

 // send the desired position to the drone
 aciUpdateCmdPacket(0);
 Sleep(1);
}
```

or like this (assuming you've defined `xdes`, `ydes`, and so forth according to the standard flight test)

```
if (1) {
 // apply the standard flight test to choose the goal position
 QuadGoal.yaw = 0.0;
 if (timer <= tdes[0]) {
  QuadGoal.Pos.x = xdes[0];
  QuadGoal.Pos.y = zdes[0];
  QuadGoal.Pos.z = ydes[0];
 } else if (timer >= tdes[ndes-1]) {
  QuadGoal.Pos.x = xdes[ndes-1];
  QuadGoal.Pos.y = zdes[ndes-1];
  QuadGoal.Pos.z = ydes[ndes-1];
 } else {
```

```
   int i;    // we have to initialize the loop variable before the loop, because
             // the compiler we're using follows the C89 standard (very old)
   for (i=1; i<ndes; i++) {
     if (timer <= tdes[i]) {
       float ratio = (timer - tdes[i-1]) / (tdes[i] - tdes[i-1]);
       QuadGoal.Pos.x = ((1.0 - ratio) * xdes[i-1]) + (ratio * xdes[i]);
       QuadGoal.Pos.y = ((1.0 - ratio) * ydes[i-1]) + (ratio * ydes[i]);
       QuadGoal.Pos.z = ((1.0 - ratio) * zdes[i-1]) + (ratio * zdes[i]);
       break;
     }
   }
 }

 // run the planner to update the desired position
 planner(QuadCurrent, ObstCurrent, &QuadDesired,  QuadGoal);

 // send the desired position to the drone
 aciUpdateCmdPacket(0);
 Sleep(1);
}
```

or whatever else you want, really - you get the idea. Choose the goal position, then run the planner to update the desired position.

You'll also want to change the `fprintf` lines so you log the goal position (`QuadGoal.Pos.x` and so forth).

Now, add this to `planner.h`:

```
```

void planner(State6_f QuadCurrent, State6_f ObstCurrent, PosYaw_f *QuadDesired, State_6_f QuadGoal);

```

It declares the `planner()` function that you just called in `main.c`.

Finally, add this to `planner.c`:

```
// Parameters (change all of these as appropriate)
float katt = 1;
float batt = 1;
float krep = 1;
float brep = 1;
float kdes = 1;
float bdes = 1;
float r = 1;   // drone radius
float s = 1;   // obstacle radius

void planner(State6_f QuadCurrent, State6_f ObstCurrent, PosYaw_f *QuadDesired, State_6_f QuadGoal)
{
   // Do not change this line
   QuadDesired->yaw = QuadGoal.yaw;

   // Replace these lines with our method of collision avoidance
   QuadDesired->Pos.x = QuadGoal.Pos.x;
   QuadDesired->Pos.y = QuadGoal.Pos.y;
   QuadDesired->Pos.z = QuadGoal.Pos.z;
}
```
```

```

It actually defines the planner function. You may want to start with the following template:

```

```
void planner(State6_f QuadCurrent, State6_f ObstCurrent, PosYaw_f *QuadDesired, State_6_f QuadGoal)
{
 // Define some useful variables (these are just examples)
 float d;          // distance to obstacle
 float dgrad[3];    // gradient of distance to obstacle
 float fgrad[3];    // gradient of potential function
 float v[3];       // you can use this to store a vector (e.g., q - qgoal, or q - p)
 float vnorm;      // you can use this to store the norm of a vector (e.g., ||q - qgoal||, or ||q - p||)
 int i;            // you can use this for a counter (e.g., in a "for" loop)

 // Initialize gradient
 fgrad[0] = 0;
 fgrad[1] = 0;
 fgrad[2] = 0;

 // Add attractive part of gradient (FIXME)
 //
 //  After some intermediate calculations, your code should
 //  do something like:
 //
 //     fgrad[0] += ... ;
 //     fgrad[1] += ... ;
 //     fgrad[2] += ... ;
 //
```

```
 // Add repulsive part of gradient (FIXME)
 //
 //  After some intermediate calculations, your code should
 //  do something like:
 //
 //     fgrad[0] += ... ;
 //     fgrad[1] += ... ;
 //     fgrad[2] += ... ;
 //


 // Take a step (FIXME)
 //
 //  After some intermediate calculations, your code should
 //  do something like:
 //
 //     QuadDesired->Pos.x += ... ;
 //     QuadDesired->Pos.y += ... ;
 //     QuadDesired->Pos.z += ... ;
 //
 //  The reason you need to use "->" instead of "." here is because a pointer
 //  to QuadDesired was passed to this function - this means that changes to
 //  QuadDesired persist (it's as if this function returned QuadDesired).
}
```

Note that this template is more or less exactly what you would have implemented in MATLAB.

Some differences between MATLAB and C

Conditional statements

A conditional statement in MATLAB:

```
if ( ... )
   % do something
else
   % do something different
end
```

A conditional statement in C:

```
if ( ... ) {
   // do something
} else {
   // do something different
}
```

Assignment

Assigning a value to a vector in MATLAB:

```
v = [3; -1; 4];
```

Assigning a value to a vector in C:

```
//
// v must have previously been declared as:
//    float v[3];
//
v[0] = 3;    // notice that indices start at 0 in C
v[1] = -1;
v[2] = 4;
```

Sums

Adding a value to a variable in MATLAB:

```
x = x + 5;
```

Adding a value to a variable in C:

```
//
// x must have previously been declared as:
//    float x;
//
x += 5;      // option #1
x = x + 5;   // option #2
```

Norms

Computing the 2-norm of a vector (i.e., its "length" or "magnitude") in MATLAB:

```
vnorm = norm(v);
```

Computing the 2-norm of a vector in C:

```
//
// v and vnorm must have previously been declared as:
//    float v[3];
//    float vnorm;
//
vnorm = sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
```