Localization & Navigation

MIT 6.270

2013 IAP

These notes were prepared for the 2013 6.270 Lectures based on the lecture slides used in the 2012 6.270 Lectures by Scott Bezek.

Overview

By now, you should already be familiar with control systems – feedback, proportional/PID controls etc. Equipped with that, we now wish to actually get the robot somewhere. In terms of navigation, we ultimately would want to command the robot say, “moveToPoint(x, y)”. But before we can do that, we must first be able to know where the robot is. This is the topic of *localization*.

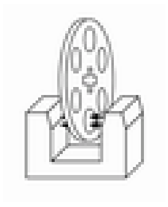
Localization

To achieve the goal of navigation, we need to be able know where the robot is in an environment at any given point in time, as accurately as possible. This is no easy task, but luckily for you guys, there is a reliable *visual positioning system*, or *VPS*, readily available. There is also the rather traditional method of dead-reckoning. In past 6.270 competitions (say, at least before 2010), most of it was done entirely by dead-reckoning, or by more tedious methods such as line following[1] and wall aligning[2].

Dead-reckoning

The Sensors

In the 6.270 kits, gyroscopes can determine heading, while a Lego pulley wheel and a breakbeam sensor arranged as below can function as a shaft encoder to determine distance travelled.



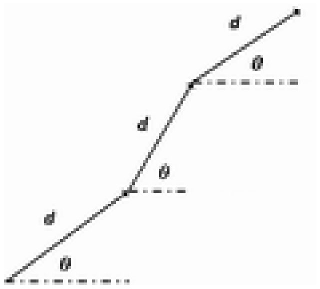
Placing one shaft encoder on each wheel and taking the average can theoretically give you the distance travelled by the robot. However, drive wheels may slip, which may lead to inaccurate deducing of distance. A solution around this is to use free wheels, located at the center of rotation of the robot. These should be as friction free as possible, so that no slipping occurs. (On a sidenote, even if you don’t use encoders for localization, they could also be useful in collision checking, so it could be a good idea to think about that.)

[1] Navigating by following a line on the ground. Sensors such as the IR-LED/Phototransistor can be used to detect the line.
[2] Navigating by moving along a wall, much like how you would get to the other side of a dark room.

Dead-Reckoning

By knowing distance travelled and heading direction, the simplest method that can be employed is *dead-reckoning*. Suppose we know that we start off at the point (x, y). Consider the following

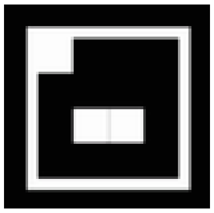
```
d = encoder_read(ENC_PORT) * CONV_FACTOR;
encoder_reset(ENC_PORT);
pos_x = pos_x + d * cos(theta); //use old heading
pos_y = pos_y + d * sin(theta);
theta = gyro_get_degrees() % 360;  //update current heading
```



By applying this every iteration, we can inductively estimate our current location. Since sensors are local, this method gives faster updates and relatively smoother and more accurate short-term estimations compared to those obtained using the VPS. However, one obvious problem with this method is that errors will accumulate and sensors will drift, thereby causing increasingly inaccurate results in the long run. In the appendix I talk about how dead-reckoning and the global positioning system can be combined to give better approximations in noisy environments.

VPS

For 6.270, we have a global positioning system, which in our case is a vision positioning system. Each robot will have a fiducial pattern placed above that an overhead camera can track at constant intervals. The VPS determines the positions of the robots, and transmits that data wirelessly to the robots. The update frequency is around 17 times per second, so usually there aren't any issues of latency. The VPS data packet will also contain necessary gaming information in the future.



How to use it

The Code

The following sets up the necessary variables.
Add the following before usetup (e.g. in definitions):

```
extern volatile uint8_t robot_id;
```

Add the following inside usetup:

```
robot_id = ***your team number***;
```

To get the newest data at any point (if it exists), call the following:
copy_objects();

The 'game.coords'' array will contain the most up-to-date position information (from the most recent call to copy_objects). Your robot's information is at index 0:

```
game.coords[0].x
game.coords[0].y
game.coords[0].theta
```

You can find out when the data was received by looking at

```
position_microtime[0]
```

You can subtract this value from get_time_us() to find out how many microseconds ago the data was received.

The Data

The x, y, and theta values are all <u>signed integers</u> that range from -2048 to 2047. (0,0) is the middle of the playing field area. A theta value of 0 points along the positive x axis and positive theta goes counterclockwise (like a standard coordinate system).

To convert from the VPS units to the corresponding proper units (assume feet for coordinates, and degrees for theta) you would need appropriate conversion factors. For angles, the conversion would be

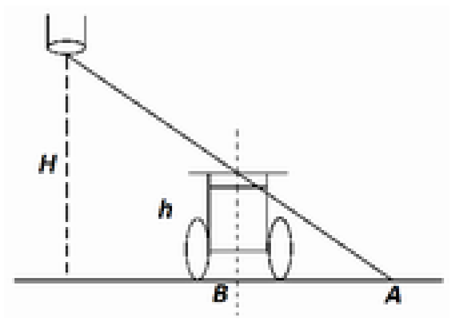
deg_angle = vps_angle / 2047 * 180.

While for distances, it depends on the dimensions of the playing field. Say, if the field stretches LxL feet, then the conversion would be

ft_coord = vps_coord / 2047 * (L / 2).

Since the field length may change from mock competition to the actual playing field, it is a good idea to define the conversion factors as constants so that they are easily updated. For example, if L = 6, then
#define VPS_PER_FOOT = 682.6666;
so that

ft_coord = vps_coord * VPS_PER_FOOT.



One final and important thing to note is that the VPS coordinates data give you the coordinates of the point A in the above diagram, when in reality you wish to know the coordinates of point B. This isn't such a difficult problem to solve though. Suppose we know that the height of the camera is H and the height of our robot is h, then basic geometry gives us

real_coord = (H - h) / H * vps_coord

Navigation

Now that we know where the robot is, navigation isn't such a difficult problem anymore. Let's think about what sort of functions and variables we will need to navigate our robot. In coding a main idea is *abstraction*, so I will try to talk a bit about the different levels of abstraction.

A Brief Note on Bottom-up/Top-down Design

There are generally two kinds of programming styles, the *bottom-up approach* and *top-down approach*. In the former, individual functions or modules are created from the basic level and linked together to form higher level functions. The top-down approach is just the opposite, where higher level functions are programmed first, with functions that will be needed supposed, then conquered inductively down the levels.

In 6.270, holistically, the approach is more often bottom-up, as we figure out the different levels of our system progressively - say, from control to navigation to task-solving to gameplay etc. The idea is that once a lower level function is written, it acts as a black box that higher level functions call. For the time being though, we are dealing with a specific subsystem, it kind of helps to think in a top-down way, as we now see.

Navigation Subsystem

So we want to build a navigation subsystem. Ultimately, we want a function that says something like moveToPoint(desired_point, velocity). What functions will we need to accomplish something like this? In the simplest form, the robot will first need to rotate to the right direction, then drive straight until it arrives at the desired point, at which it stops.

```
- float determineHeading(current_point, desired_point)
- void rotateToHeading(float desired_heading) //rotate to heading
- void moveStraight(desired_heading, velocity) //PID controller
- float distanceTo(current_point, desired_point)
- void brake()
- void update() //update current position
```

It also helps to keep track of states in gameplay, so we might also want boolean (stored as ints in C) "state variables" such as rotation_state, moving_state, etc.

So our moveToPoint function could roughly look something like (in semi-pseudocode)

```
moveToPoint(desired_point, velocity) {
        update();
        desired_heading = determineHeading(current_point, desired_point);
        rotateToHeading(desired_heading);

        while(distanceTo(current_point, desired_point) < TOLERANCE) {
                moveStraight(desired_heading, velocity);
        }

        brake();
}
```

Now let's consider the functions we used. Things like determineHeading and distanceTo are basic math, so no subfunctions are needed. rotateToHeading and moveStraight were discussed in control systems, where we had subfunctions like setVelocity(velocity) and limitVelocity(). For simplicity, let's assume a proportional controller

```
int motor_left_vel, motor_right_vel;
moveStraight(desired_heading, velocity) {
        update();
        moving_state = 1;
        motor_left_vel += FACTOR * delta_heading;
        motor_right_vel -= FACTOR * delta_heading;
        setMotorVelocity(motor_left_vel, motor_right_vel);

}
```

where setMotorVelocity does what the name suggests. Such a hierarchical build-up shows the different levels of abstraction in this subsystem. Later on, when it comes to gameplay, abstraction becomes increasingly more important and hopefully clearer.

Tidbits

The idea of using states can also be applied to using threads, so that the robot has a separate thread for navigation. My team did not employ such a method, but the example given the 2012 lecture notes does. Feel free to check that out if you are considering using threads.

In the long run, flexibility is a pretty important issue to think about. So it helps to think about how you might use your navigation subsystem in your hierarchical program so that you can code accordingly.
In particular, the navigation system should be able to incorporate collision checking and timeout mechanisms.

Conclusion

So that's pretty much about it for localization and navigation. Hopefully at this point, with some work you can be able to program your robot to do some basic navigation. Be warned though that calibration may take a chunk of time.

I included some random tips below just in case anyone would be interested. I also included an appendix containing some material on more general localization methods. It was taught last year, but with the robust VPS not much of it was applicable. Nonetheless, I found the material pretty cool, and in fact met a lot of the ideas in classes and projects later, so thought I should discuss a bit about it.

Appendix

The VPS system we currently have is reliable enough for our purposes, and is generally the way to go for localization for 6.270. However, in "real-life", positioning systems most likely aren't so reliable - there is generally a fair degree of latency and noise in the data received. Data received may jitter and packets may be dropped. For example, think about the GPS when you drive the car or use the phone. In these cases, we need other methods to arrive at better estimations of position.

Merging VPS with Dead-reckoning

So how can we combine these two methods so that we can arrive at better approximations? The main two issues we have with the VPS data are potential jitters and latency. While dead-reckoning does a good job in short-term approximations.

For potential jitters, a possible idea is to think in terms of confidence. It would be reasonable to say that the faster the robot is going, the more likely the data we receive could jitter and be not so accurate. Thus, we could define a confidence factor based on velocity. For example, if (x, y) is currently based on dead-reckoning, we could update it by

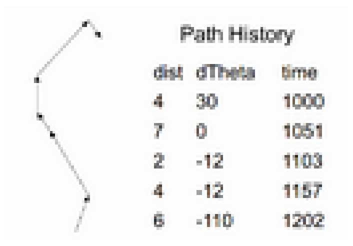
```
        confidence = (255 – abs(motor_vel)) / 255.0;
        x = confidence * vps_data.x + (1 - confidence) * x;
        y = confidence * vps_data.y + (1 - confidence) * y;
```

to put VPS into consideration.

However, this method still does not solve the problem of latency. By latency we mean that the data we receive at a given point in time corresponds to the state of the robot a certain time point beforehand. If say the VPS has a latency of 300ms and we receive VPS data giving us coordinates (x, y), then it is actually telling us that 300ms ago the robot was at (x, y).

With a relatively accurate point to start with 300ms ago, and with dead-reckoning relatively accurate in short periods, it makes sense to apply dead-reckoning to the VPS data to approximate the current position better. In order to do this, we would need to keep track of the path history of the robot. That way, each time we receive new VPS data, we can recreate our path from the time the data actually represents and obtain a closer approximation of our current location. An example combining this method and the confidence factor idea is illustrated below.

```
    confidence = (255 – abs(motor_vel)) / 255.0
    data_time = vps_data.timestamp – 300  // 300ms latency
    dx_since_data = get_total_dx_since(data_time)
    dy_since_data = get_total_dy_since(data_time)
        // dead-reckoning
    vps_x = vps_data.x + dx_since_data
```



Path History

dist	dTheta	time
4	30	1000
7	0	1051
2	-12	1103
4	-12	1157
6	-110	1202

```
vps_y = vps_data.y + dy_since_data
x = confidence*vps_x + (1-confidence)*x
y = confidence*vps_y + (1-confidence)*y
```

The ideas mentioned above are just a glimpse of what you might see in real-life engineering situations which I found pretty cool. There are of course many other ways to consider this problem, each with their own advantages and disadvantages.