ZERO ROBOTICS

SPHERES ISS CHALLENGE

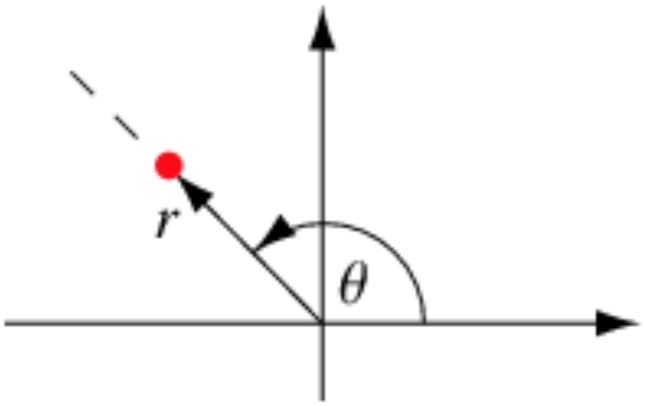
Revolving: Polar Coordinates

Polar Coordinates

- This tutorial will teach you how to program your satellite to revolve using **polar coordinates**.
- Using polar coordinates to revolve allows your satellite to maintain a constant radius of orbit, as well as adhere to a target angular velocity.
- However, using polar coordinates has its limitations. This tutorial will only handle revolving in 2-D about the origin.

Polar Coordinates

- The 2-D polar coordinate system is based on **radius** (**r**) and **angle** (θ).



- Its relation to the Cartesian coordinate system is below.

$$x = \mathbf{r}\cos\theta$$
$$y = \mathbf{r}\sin\theta$$

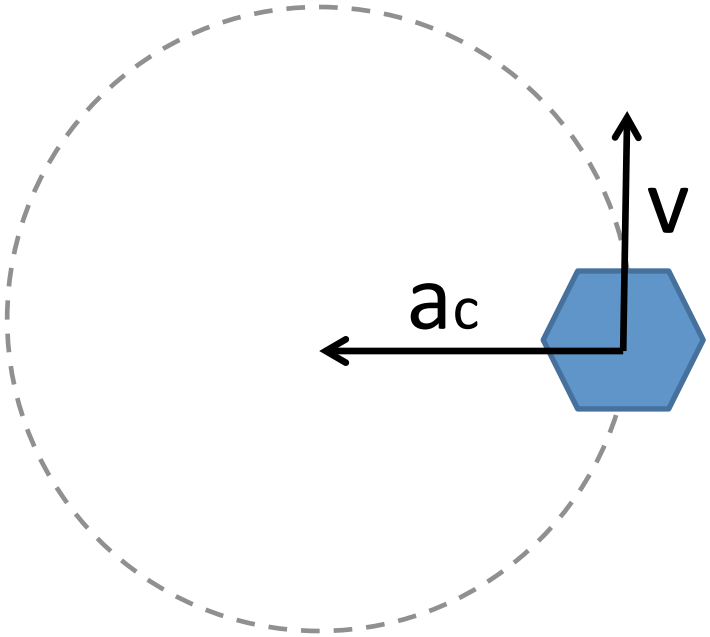
Image via Wolfram MathWorld

Uniform Circular Motion

- **Uniform circular motion** is revolution around a point while maintaining a constant speed and radius. For the best and most stable path, always try to achieve uniform circular motion.
- Although speed is constant, velocity is always changing because it is tangent to the path of revolution
- The acceleration vector always points toward the center of rotation. This type of acceleration is called **centripetal acceleration**. Centripetal acceleration is dependent on tangential velocity and the radius of the orbit.

$$a_c = \frac{v^2}{r}$$

Uniform Circular Motion



The Scenario

- Revolving with polar coordinates is simple. Choose a radius and a target angular speed, calculate x and y, and set your position target.
- Let's do an example. We want to revolve around a vertical axis at the origin, keeping a radius of 0.3 m and a speed of 2 deg/sec.
- The technique is simple, but there will be a significant amount of coding in this tutorial. When you program your satellite for the competition, you may find yourself writing hundreds of lines of code. This algorithm isn't nearly that long, but it's good practice.

Prep Work

- Create a new project called Project20.
- Let's start by creating all our variables.

Variable	Purpose
axis[3]	Axis of rotation
targetPos[3]	Store our x and y targets
targetRadius	Polar coordinate r
angle	Polar coordinate θ
myState	State array
myPos	Current position
targetAtt	Face the axis of rotation
actualRadius	Debug radius

Initialize Variables

- Only a few variables need to be initialized.
- Set angle to 0.
- Set targetRadius to 0.3.
- Initialize axis with 0's because we are rotating about the origin.

```
11 void init(){
12   angle=0.0;
13   targetRadius=0.3;
14   axis[0]=0.0;
15   axis[1]=0.0;
16   axis[2]=0.0;
17 }
```

Target Position

- We need to use the equations x = **r**cosθ and y = **r**sinθ. Since we are revolving in 2-D, our z-component is 0.
- Before you see the code below, see if you can figure it out yourself. Use the trig functions in math.h in the API.
- After we set the target position, we need to increment angle by 2° so we can maintain our target speed. Remember that loop() is called every second.

```
19 void loop(){
20   targetPos[0]=targetRadius*cosf(angle);
21   targetPos[1]=targetRadius*sinf(angle);
22   targetPos[2]=0.0;
23   api.setPositionTarget(targetPos);
24   angle+=2.0*PI/180;
```

Attitude Target

- To maintain uniform circular motion, the satellite must face the center of rotation at all times. We can accomplish this by finding and setting an attitude target.
  - If this sounds unfamiliar, walk through the tutorial setAttitudeTarget, Revisited.
- Start by retrieving myState and writing the first three elements to myPos.
- We want targetAtt to point from myPos to axis, so use mathVecSubtract.

```
26    api.getMyZRState(myState);
27    for (int i=0; i<3; i++)
28      myPos[i]=myState[i];
29    mathVecSubtract(targetAtt,axis,myPos,3);
```

Radius

- Right now, targetAtt represents the actual radius of the circular path. Before we make it a unit vector, we want to store the magnitude in actualRadius for debugging purposes.

```
32    actualRadius=mathVecMagnitude(targetAtt,3);
33    DEBUG(("%f",actualRadius));
```

- Now we can normalize targetAtt and set our attitude target.

```
34    mathVecNormalize(targetAtt,3);
35    api.setAttitudeTarget(targetAtt);
36 }
```

- Compile and run.

Results

- Keep an eye on the console. Once the satellite enters orbit, the radius is maintained at approximately 0.3 m.
- Also observe the state array values in the top left hand corner. Once the satellite enters orbit, the y-component of angular velocity hovers around 2 deg/sec.
- The main issue is the inefficient maneuver the satellite performs before it begins revolving. This is the result of a poorly-placed entry point, more commonly known as a **waypoint**.

Waypoint

- We want you to write unique algorithms. We won't teach you how to set a waypoint because you have the necessary skills. But, we will give you a few hints.
- Only set a waypoint *before* you are in orbit. Find a way to decide if your satellite is revolving, and use a boolean to make sure you don't accidentally call the waypoint while in orbit. Conditionals will be necessary.
  - A boolean is a variable that is either true or false.
- Use momentum to your advantage. In this example, we had to completely stop and move in the opposite direction to revolve. Make sure this NEVER happens.
- Always aim for efficiency. Think outside the box!