Sensor System

Background

The submarine currently has 7 different types of sensors: accelerometer, gyroscope, magnetometer, depth sensor, battery voltage sensor, hydrophones, and the start switch.

Raw Sensors

3-Axis Sensors

The accelerometer, gyroscope, and magnetometer are 3-axis sensors, meaning they report X, Y, and Z values. The coordinate system is the same as the one defined for movement. See the Movement section for details. Positive X is to the left of the sub, positive Y is to the front, and positive Z is down. Because the sensors may not be mounted in the proper orientation, translation is needed. This shall happen in the lowest level (`sensor_microcontroller_interface.py`), so all other computer modules can assume the sensors are orientated correctly.

Accelerometer

- datasheet
- Measures acceleration in the X, Y, and Z axes
- This includes the force of acceleration due to gravity
- if the sub is stationary and sitting upright, you should expect a (relatively) large value on the Z axis, and close to zero on the X and Y axes
- if the sub is diving, the Z value should be less than when stationary (and would approach zero if the sub was in a free-fall
- if the sub is rising, the Z value should be greater than when stationary
- The units are such that a value of ~256 on the sensor is equal to the force of gravity (1 g, or 9.81 m/s^2).
- Therefore, wit the sub sitting stationary we would expect a value of ~256 on the Z-axis of the sensor

Gyroscope

- datasheet
- Measures angular velocity in the X, Y, and Z axes
- rotation around the Z-axis is yaw, Y-axis is roll, X-axis is pitch
- If the sub is sitting perfectly still, we should see values close to zero on all axes
- The units are such that a value of ~14.375 on the sensor is equal to a rotation speed of 1 degree/second

Magnetometer

- datasheet
- Measures magnetic field strength in the X, Y, and Z axes
- This will **not** just give us an angle of degrees from north, post-processing is needed
- The units are such that a value of ~10 on the sensor is equal to 1 µT
- According to the datasheet, we should expect a magnetic field strength due to the earth of approximately 50 µT. This would be equal to a value of about 500 on the sensor. (This has not been tested)

ADC Based Sensors

The depth sensor and the battery voltage sensors are ADC based. This means that an analog voltage is being presented to the microcontroller, and the microcontroller is making a digital approximation of that value. The resolution of the ADC is 10 bits, so values between 0-1023 are possible.

Depth Sensor

- Measures the submarine's depth
- Value is linearly related to the depth
- the maximum resolution of this sensor is about 0.1728 feet.

Equation is:

> depth_feet = (sensor_value * 0.1728) - 10.341

Battery Voltage Sensor

- Measures the battery's voltage
- Value is linearly related to the voltage
- The maximum resolution of this sensor is about 25 mV

Equation is:

> actual_voltage = sensor_value * (3.3/1024*7.5)

GPIO Sensors

These sensors use the GPIO pins of the microcontroller. They can detect a high value, low, value, or a changing value. The only GPIO sensor on the submarine right now is the mission start switch.

Mission Start Switch

- The microcontroller tells the computer when it sees a rising edge on the start switch
- In the field, it has been observed that the switch contacts bounce when we flip the switch, resulting the in the microcontroller seeing multiple rising edges and therefore telling the

computer the switch has been flipped multiple times, even when the switch has only been flipped once. It appears to bounce ~3 times each time it is flipped.

Hydrophones

The hydrophones are technically ADC sensors, but they deserve their own section because they are not accessed using the microcontroller. Instead, the audio jacks on the computer are used as high-speed ADCs. Custom software has been written to read the hydrophones and compute an approximate angle for where the hydrophone is.

Processed Sensors

`insert description of data available to the submarine after raw sensors have been processed`

Update for 2016 competition:

The submarine currently has several types of sensors:

- 4x 3-axis Gyroscope
- 4x 3-axis Accelerometer
- 4x 3-axis Magnetometer
- 4x 2mm precision Depth Sensors
- 4x hydrophones (partially implemented)
- 8x Blue Robotics T-200 Thrusters w/ embedded sensors: Voltage, Current, and RPM
- 2x Forward Stero Cameras
- 1x Bottom facing camera (partially implemented)
- Individual Battery Cell Voltage monitoring
- GPIO board (not implemented)

The first section will outline how 3-D rotation is managed in our system. This is not the only method: Euler angles and Quaternions are equally valid approaches to 3D rotation, however they have some drawbacks that can make them undesirable

The following section will explain the basics surrounding each sensor, how that data is transformed into relevant information, how the sensor readings can be distorted and how to correct them, and how things can be optimized.

The Final section will include the basics of Particle Filtering and Kalman filtering, which are data-fusion techniques that seek to create optimal estimates of state variables given measurements, a kinematic model, and how noisy each of those are.

3D Rotation:

A 3D rotation is the multiplication of a vector, or a group of vectors, by a 3×3 matrix with row and column magnitudes of 1, and a determinate of 1. A rotation matrix is Orthogonal, meaning that it's transpose is also its inverse - a property greatly exploited with fusing and correcting sensor data. The transformation defined by this matrix is a pure rotation with no stretching or compressing along any

axis. Standard form is for the rotation matrix R to be left-multiplied by column vectors [X;Y;Z]

R x [x y z]' = [rx ry rz]'

R x [x1 y1 z1; x2 y2 z2;... xn yn zn]' = [rx1 ry1 rz1; rx2 ry2 rz2;... rxn ryn rzn]'

Rotation in our system is made up of Yaw, Pitch, and Roll. To arrive at an arbitrary rotation specified by these values, begin at the unrotated state where the submarine's intrinsic XYZ axes align with the global XYZ axes. A rotation is then defined as a yaw, then pitch, then roll, around the original z axis, the yawed y axis, and then yawed-pitched x axis respectively. These are right-hand rotations, so positive yaw is to the left, positive pitch is down (cry moar) and positive roll is clockwise.

`<insert equation of rotating about arbitrary axis xyz  <insert long-form r3D tracking Y and X axes>`

Try it out. Like it or not, getting the hang of these rotations is crucial to everything that follows.

Let's try an example. Lets do a rotation: yaw:15 degrees, pitch:-60 degrees, roll:135 degrees

First stick your right arm straight out.

Now point your entire torso 15 degrees to the left.

Now point your arm up 60 degrees about your shoulder.

Now quickly roll your forearm over 135 degrees, so you look like an Opera Singer instead of a Nazi.

Let's try another. yaw: -30 pitch:-40, roll -70

Weird orientation? You bet. But it's a possible arbitrary one, and our rotation engine can do it.

Lets try a third one. yaw 180, pitch 0, roll 180.

Now try a fourth one. yaw 0, pitch 180, roll 0.

As you'll notice, three and four are the same rotation, with different roll/pitch/yaw values. While the 3×3 rotation matrix those commands produce will be identical, you should take note that different rotation paths can lead to the same place. Nonlinear and sucky, I know. It can't help it. 3D stuff is weird. Rotation matrices really DO try to help you where-ever they can. Cut them some slack.

Brief aside:

Why do we want to yaw,pitch,and roll based on our current position rather than the global axes? There are a number of reasons, but the simplest is that for controlling the submarine, covered in the Control Section, the submarine has thrusters typically mounted such that they can perform pure rolls, pitches, and yaws. And these thrusters rotate with the submarine. So if I tell the submarine to yaw while it's tilted on its side, it's going to rotate along the XZ plane, not the XY plane. Setting up our rotation engine this way lets us use intuition most effectively for rotation.

Euler angles, as an example, constitute a yaw, followed by a roll about that new location, followed by another yaw. If you play Kerbal Space Program, you can imagine how this system would be useful for defining a location about a planet. Yaw to where your orbit intersects Kerbal's equator, roll to your orbit's inclination, and then yaw again until you're at the specified point along that orbit. But that's hardly useful to us controlling a submarine, or even a spaceship in KSP. In fact, you'll find that

controlling our submarine is most akin to controlling a space ship compared with any other vehicle.

/Tangent over

In matrix operations, order of multiplication matters. RX = X* but XR != X* And we generally think of the order of multiplication as occuring from right to left.

So in the system ABX = Y, (AB)X=Y is true, but it is generally thought of as A(BX) = Y. The A transform is operating on the outcome of the B transform on X. This is purely semantic, but it is a good way to structure the order in your head so you don't try to do BAX by mistake.

But consider rotations. If we want to take our system and string together two rotations, R1 and R2, how would we go about it?

R2*R1*P = rP ? We just perform the R2 rotation on the result of R1*P, right?

In fact no. As nice as this would be, let's think this through.

Consider what would occur in the R2*R1*X system. Imagine that R1 is a 45 degree yaw, 0 degree pitch, 0 degree roll rotation. And assume that R2 is a 0 degree yaw, 90 degree pitch, 0 degree roll rotation. Lets assume that the two points we're rotating are represented by the vectors P1 = [1 0 0]' and P2 = [0 1 0]'

Go ahead and put your right arm out in front of you, and your left arm to the side, and execute the two rotations. You should end with your right arm pointed up, and your left arm pointing back-left. But does this happen mathematically?

R1*P would rotation our point P1 from [1 0 0]' to [.707 .707 0]' and point P2 from [0 1 0]' to [-.707 .707 0] - a simple 45 degree yaw to the left. Next we'd like that point to be pitched up 90 degrees, so that P1 is at location [0 0 1], and P2 remains at its location. In short, we'd like to roll by 90 degree about the vector the represent P2 - our intrinsic y-axis.

R1*P = r1P = [.707 .707 0; -.707 .707 0]'

But if we used R2 = r3D([0 90 0]) to generate our R2 matrix, how would it know that we want it to pitch around this 45-degree-yawed y-axis? It doesn't know that - it assumes that there is 0 yaw, so it performs a rotation round the GLOBAL y-axis.

What's going to happen to R2*r1P? Pitch those two points P1 and P2 about the global Y axis by 90 degrees. P1 will get rotate up to [0 .707 .707] and P2 will get rotated down to [0 .707 -.707]. That isn't where we wanted to end up at all!

So how do we string together rotations? We reverse the process.

First pitch P1 and P2 up by 90 degrees. P1 = [0 0 1]' and P2 = [0 1 0]'. THEN yaw P1 and P2, not around the plane made by P1 and P2, but by a new rotation rotating around the global z-axis. P1 is on that axis, so it remains at P1 = [0 0 1]'. P2 gets rotated from sitting along the global y axis, to being at location P2 = [-.707 .707 0]. Exactly where we wanted them!

So we find that, to repeatedly yaw1-pitch1-roll1-yaw2-pitch2-roll2 about our moving, intrinsic axis, we have to [yaw2-pitch2-roll2]-[yaw1-pitch1-roll1]. You can imagine it like building up a stack. First rotation is perform last, with all following rotations stacked on top of it.

Try rotation example 2 from above again. yaw: -30, pitch -40, roll -70.

This time, stick out your arm straight ahead. Then roll your forearm, then pitch your shoulder, then rotate your torso.

You should be at the same place as doing things in the reverse order, but you'll find that all of your rotations were done in reverse order, and all being performed about the original X, Y, and Z axes because they weren't yet modified.

Now, let's take another example, and apply the above, to find out something nice. Now lets do 3 rotations. The first a pure yaw, the second a pure pitch, and the third a pure roll.

Our R matrix will have to be R1*R2*R3 = R. Or [yaw3-pitch3-roll3] then [yaw2-pitch2-roll2] then [yaw1-pitch1-roll1] tracking the intrinsic axes.

Y1P1R1*Y2P2R2*Y3P3R3Y1P1R1 = Y1II*P2I*IIR3* = Y1*P2*R3.

Well look at that, doing a yaw, then a pitch, then a roll, around our tracked intrinsic axes, is equivalent to doing the roll first about the origin, followed by pitching about the origin, followed by yawing about the origin. This means, among other things, that r3D can be written FAR more efficiently.

<r3D revised:>

This is also important from another avenue. Lets say I do two rotations. Now what's our current yaw, pitch, and roll position? It's not the sum of the two yaws and pitches and rolls. If I do two rotations, a yaw and roll, and then a pitch, where am I located? Pitching from that rolled orientation, versus a normal one, means that some of my pitching is going to add to my yaw at the cost of gaining true pitch. If I yawed and rolled 45 degrees, and then pitched -90 degrees, I'd be sitting above the global Y axis (yaw = 90) at a pitch of only -45 degrees, and a roll of 45 degrees. Likewise, if I yawed, then rolled, then yawed by 90 degrees, I'd be sitting below the global Y axis (yaw 90 degrees) and would have somehow gotten 45 degrees of pitch out of the bargain.

Let's say we do a big string of rotations. Say 10 random rotations in a row. Easy, that's R1*R2*R3...R10. But where are we now? While there are many rotational routes you can take to arrive at a position, we NEED to be able to quickly determine WHERE that final position is just based off of the resultant Rotation matrix R = R1*R2*...*R10.

Where ever we end up pointing can be described by the single yaw, pitch, roll combination it takes to get there. So what must that equivalent matrix be?

Yaw*Pitch*Roll = R

Our r3D function is now simple enough, because these rotations are around the global XYZ axes, that we can follow through what each of the 9 elements of R equals as a function of the roll, pitch, and yaw values.

<track out pure yaw, pitch, and roll rotations mutliplied together>

Okay, not so bad, right? If we look to the bottom left, we find that no matter what, R(3,3) is the negative sine of our pitch. So pitch = asind(R(3,3)).

And now look at the first two elements in the first column. cosine and sine of yaw multiplied by the

cosine of pitch. We could take our newly-found pitch value, take its cosine, divide it away from either element, take the arcsine or arccos, and get yaw. But there is a more elegant way - and one better in the case where the elements might have some noise. What is R(2,1) / R(1,1)? The cosine(pitch) term divides out, and we're left with sin(yaw)/cos(yaw) = tan(yaw). So yaw is arctan2d(R(2,1),R(1,1)).

A similar trick with the third row gets you the roll component from the arctan2d(R(3,3),R(3,2))

So we get ir3D:
<code for ir3D>

Some enterprising people might notice that there's a fatal issue, when pitch is exactly +/- 90 degrees. When that occurs, the other terms in the first column and third row are both zero. So ir3D will return a yaw and pitch of zero degrees, regardless of what yawing and rolling has taken place. This is a problem, because in the case of a 90 degree pitch, the sum of roll and yaw is what dictates the final value. We'd have to use those messier top-right terms to divine their sum.

In practice, where we are using perfectly constructed but arbitrarily-valued rotations, this is never a problem. If the pitch is slightly off of +/-90 then some non-zero values will occupy the other elements and their ratios will properly reflect the roll and yaw that would reach that rotation. However, if we are using somewhat imperfect rotations, say an R matrix not produced by r3D but solved for by noisy sensor data, or perfectly valued rotations like we might get in the simulator, we'll have to use other methods. But this ir3D is fine for now.

So now we can produce an arbitrary rotation matrix, we know how to properly integrate together rotation matrices, and we know how to invert a 3×3 rotation matrix into a single roll-pitch-yaw combo.

One other thing to mention before moving on - the integration of rotations. Say I'm yawing and rolling at 10 degrees per second. Where am I after 1 second? yaw:10 pitch:0 roll:10 ? Hah. If only.

R != r3D([dR dP dY])

instead, R = r3D(1/n*[dR dP dY]) ^ n as the limit of n approaches infinity. You're continuously rolling and yawing from new orientations. If you kept it up long enough, you'd actually rotate until you're pointed straight-up and backwards. Essentially, you're rolling about the arbitrary axis defined by [.707 0 .707]

<fill in later how to find how rotational velocity translates to continuous rotation around an arbitrary vector>

Now back to Sensors:

Gyroscope:

Gyroscopes provide very precise measurements of rotational velocity. By precise, I mean we're talking resolution of ~1/14 degrees/second with only a one or two bit jitter. It is not necessarily accurate, however, because gyroscopes zero-value can drift over time, as a function of temperature and other physical parameters. A change of one bit will consistently reflect a change of rotational velocity about an axis, but a specific sensor value sampled at two different times will not correspond to the same rate.

To fix this, simply leave the gyroscope perfectly still for a fraction of a second, to several seconds, and take the average of the readings. This should reflect what the sensor is reading when rotation is zero.

Subtract this value from future readings, and scale by the bit resolution to find true yaw, pitch, and roll.

Depth Sensors:

tbc

ir3D:
```
<insert function here>
```