Hands-on Lab

WhIP: Wheeled Inverted Pendulum

WhIP is an NXT-based 2-wheeled inverted pendulum that is analogous to the Segway transporter. WhIP applies PID control to maintain balance. A HiTechnic NXT gyro is used to measure the body's state (angular velocity and calculated angle). Wheel state (position and velocity) is then actuated to counter-act changes in the body angle. The next effect is that there are 2 degrees-of-freedom (body and wheel positions and velocities) which are controlled to maintain balance.

Preamble: Gyro Bias and Compensation

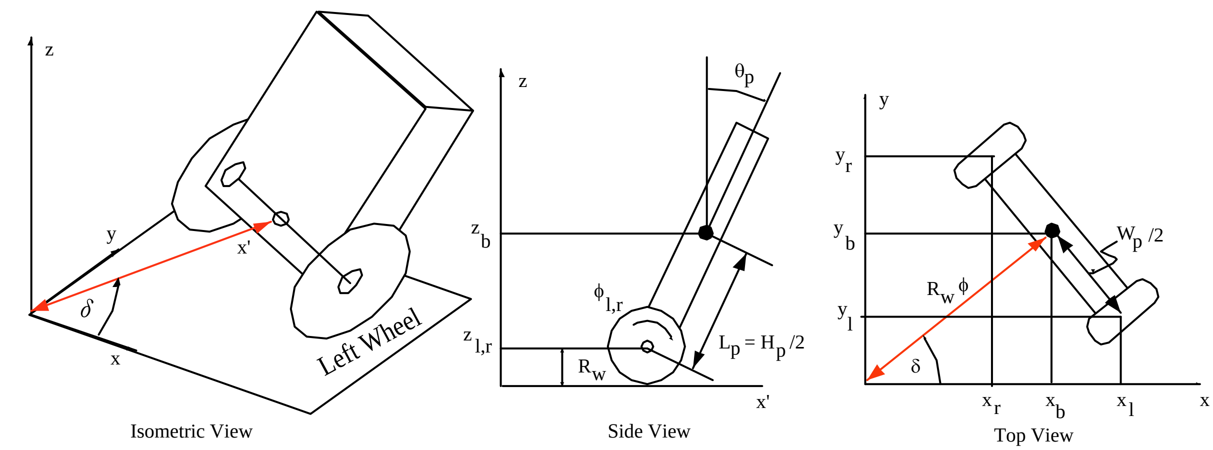

Figure: WhIP coordinates and degrees-of-freedom

Gyros measure angular velocities. These velocities can be integrated to compute angles. Thus a gyro mounted on the body can serve to provide θ (body angle) and $\dot{\theta}$ (body angular velocity). Computing an angle by integrating gyro data is notoriously subject to drift. Failure to compensate for drift will result in θ growing over time.

Step 1: Calibration – computing gyro offset

Refer to program `whipGyroCali1_0.nxc`. HiTechnic's documentation says that their NXT gyro must be calibrated to compute an offset value. Computations are achieved by sampling the gyro and then calculating the average.

```
sumOfAllRawOmegaReadings = 0.0; // zero because haven't added readings yet
totalCounts = 0; // counts number of times gyro is read
curTick = CurrentTick();  // start timer
while (CurrentTick() < (3000 + curTick)) {
    rawOmega = SensorRaw(GYROPORT); // HiTechnic gyro returns long
    Wait(150);
    totalCounts = totalCounts + 1;
               sumOfAllRawOmegaReadings = sumOfAllRawOmegaReadings + rawOmega;
               PlayTone(TONE_B7, 5); // 5 ms chirp
 }
omegaBias = sumOfAllRawOmegaReadings / totalCounts;
```

Step 2: Lay WhIP flat (and hence motionless) and execute `whipGyroCali1_0.nxc`

> **Observation:** The WhIP is motionless (i.e. gyro is stationary) but running `whipGyroCali1_0.nxc` shows that integration of the gyro's angular velocity measurements yields θ increasing (i.e. drift).

Step 3: Rename `whipGyroCali1_0.nxc to whipGyroLowPass1_0.nxc` and save. Implement a low-pass filter as follows:

```
#define LOWPASSFILTER 0.005 // constant for low-pass filter
// Value should be less than 1.0.  Small values mean that previous value
// of omegaBias (i.e. gyro bias) is weighted more.

// adjust gyro bias due to drift
     omegaBias = rawOmega*LOWPASSFILTER + (1.0-LOWPASSFILTER)*omegaBias;
     bodyOmega = rawOmega-omegaBias; // [deg/s]
     intOmegaBias = omegaBias;
```

Step 4: Execute `whipGyroLowPass1_0.nxc` and observe the resulting angle measurements

> **Observation:** A low-pass filter blocks high frequencies. The expectation is that gyro values will not change radically during run-time. As such, the *compensated* signal is a weighted sum of old values (which should not change much) and the newly acquired incoming signal.

Concept 1: WhIP PID control

Step 1: Download `whip112612.nxc`, compile and execute it. In brief, the program does the following:

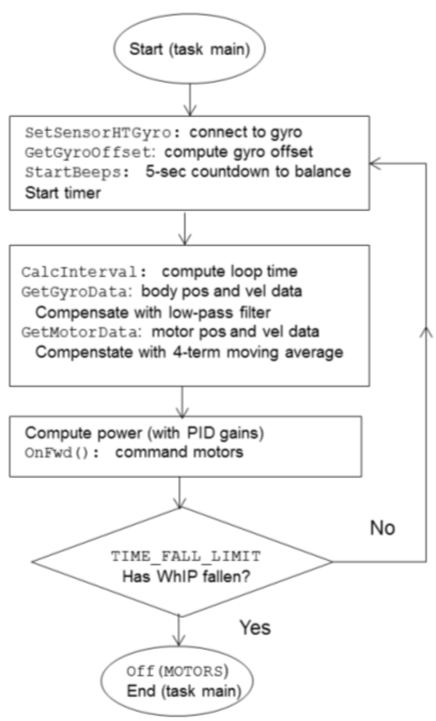

Figure 1-1: WhIP flowchart

Exercises: There are 4 gains: KGYROSPEED (body proportional gain), KGYROANGLE (body integration gain), KPOS (motor position gain), and KSPEED (motor derivative gain)

1-1 Set KGYROANGLE to 0 (keeping all other gains fixed at their default values). Note observations: does WhIP translate much? Does WhIP shake a lot? Increase KGYROANGLE and note observations.

1-2 Set KGYROSPEED to 0 (keeping all other gains fixed at their default values). Note observations: does WhIP translate much? Does WhIP shake a lot? Increase KGYROSPEED and note observations.

1-3 Set KPOS to 0 (keeping all other gains fixed at their default values). Note observations: does WhIP translate much? Does WhIP shake a lot? Increase KPOS and note observations.

1-4 Set KSPEED to 0 (keeping all other gains fixed at their default values). Note observations: does WhIP translate much? Does WhIP shake a lot? Increase KSPEED and note observations.

Concept 2: Multi-Threading

Multi-threading allows multiple processes to execute simultaneously. In previous lab programs, there was a single process – it was called `main()`. NXC provides the ability to add processes, so that they run at the same time as `main()`. The function `Follows()` is used for this purpose.

Simultaneous execution of processes is often desired so that they don't bog down each other. For example, one can create processes called `whipBalance()` and `ultrasoundResponse()`. The former keeps the WhIP balanced, while the latter polls the ultrasonic sensor.

Step 1: Mount an ultrasonic sensor on your WhIP. Connect to Port 4.

Step 2: Download `whip102315.nxc`, compile and execute it.

Note that `main()` has far fewer statements than `whip112612.nxc`. Much of the code, especially the PID components have been moved to a process called `task whipBalance().` Also, a process called `task ultrasoundResponse()`has been added. This process polls the ultrasonic sensor and plays a tone if the sensor detects objects within a specific range.

Once main ends, then the processes `whipBalance()` and `ultrasoundResponse()` commence. The processes run endlessly until the WhIP falls down. After falling down, the NXC function `StopAllTasks()` is called to kill all running tasks, and thus shuts down the motors.

Exercises: Mount an ultrasonic sensor on your WhIP and

2-1 Examine `task ultrasoundResponse()`. Play different tones based on distance. For example, play `TONE_B3` for `0 <= usSensorValue <= 30`; and `TONE_A3` for `31 < = usSensorValue <= 60`.