Elecanisms Mini Project 2 Report

Caitlin and Kyle

2/24/2015/I

Overview:

To simulate multiple virtual environments on our haptic joystick, we implemented a Finite Stat Machine (FSM) whose state is controlled over vendor specific usb connection. This allows us to change the environment and parameters from the computer without needing to reset or reprogram the microcontroller. We had to slightly modify the vendor requests provided with Brad's example code in order to meet our needs.

We then developed a command line interface using python that allows the user to communicate easily with the joystick. Our interface includes basic error checking, as well as help if you don't know the usable commands. See the video linked below for a demo.

We developed code for controlling torque, since this is the basis for most of our environments. We did this by measuring analog voltage on A0, which is proportional to current through the motor (this is the amplified voltage drop across a current sense resistor). We then implemented a very bare bones feedback loop that alters the voltage to the motor until desired torque equals actual torque. A future improvement could be to replace this feedback loop (essentially just an integrator) with a full blown PID loop. Due to the high calculation frequency relative to the operational frequency of the system, it wasn't necessary for this lab.

The environments we simulated were a spring, a damper, a rough texture, and a wall. See the video linked below for a demo of the different modes.

Important code: The following is our main function, and outlines how we multitask the functions of printing data and updating the motor (using timer interrupts), and checking the USB interface for new commands (using the main loop)

```
int16_t main(void) {
    init_clock();
    init_uart();
    init_ui();
    init_timer();
    init_pin();
    init_oc();

    Config_Motor_Pins(); // set up motor driver shield
    init_variables();

    InitUSB();
    while (USB_USWSTAT!=CONFIG_STATE) {     // while the peripheral is not configured...
        ServiceUSB();                       // ...service USB requests
    }

    timer_every(&timer3,.0005, Update_status); // check the state of the FSM, then check
    // the position of the motor, then update the voltage to the motor (PWM) based on
    // what state it is in and what it should be doing.

    timer_every(&timer1,.01,printData);

    while(1) {
        ServiceUSB(); // check to see if we need to enter
    }
}
```

The following is our method of implementing an FSM using a simple switch statement. See virtualenvironments.c for full code

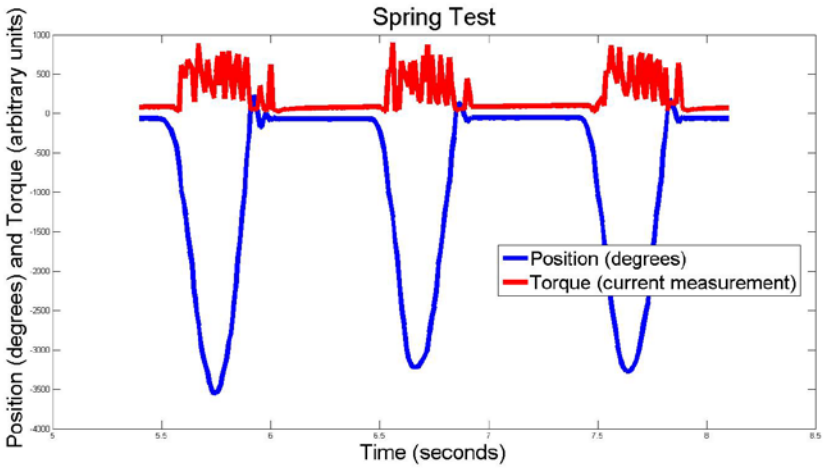
```
void Update_status(_TIMER *self){
    updatePosition(); // check the magnetoresistive sensor to make sure we always keep track of where
    // we are
    switch(command) {
        case SPRING:
            led_on(&led1); led_off(&led2); led_off(&led3); // for visual feedback (debugging).
            force_desired = -(current_position) >> 6; // arbitrary units. will fix later
            force_desired = force_desired * 15;
            set_torque();
            break;
        case DAMPED:
            led_on(&led1); led_off(&led2); led_off(&led3); // for visual feedback (debugging).
            force_desired = last_position - current_position;
            force_desired = force_desired * 15;
            set_torque();
            break;
        case TEXTURE:
            if ((current_position % 600) > 300) {
                force_desired = -(current_position) >> 6;
                force_desired = force_desired * 8;
                set_torque();
            } else {
                stop_motor();
            }
            break;
        case WALL:
            led_on(&led1); led_on(&led2); led_on(&led3); // for visual feedback (debugging).
            if (current_position > (initPos+2000)) {
                force_desired = -(current_position) >> 6;
                force_desired = force_desired * 15;
                set_torque();
            } else if (current_position < (initPos-2000)) {
                force_desired = -(current_position) >> 6;
                force_desired = force_desired * 15;
                set_torque();
            } else {
                stop_motor();
            }
            break;
        default :
            stop_motor();
            break;
    }

}
```

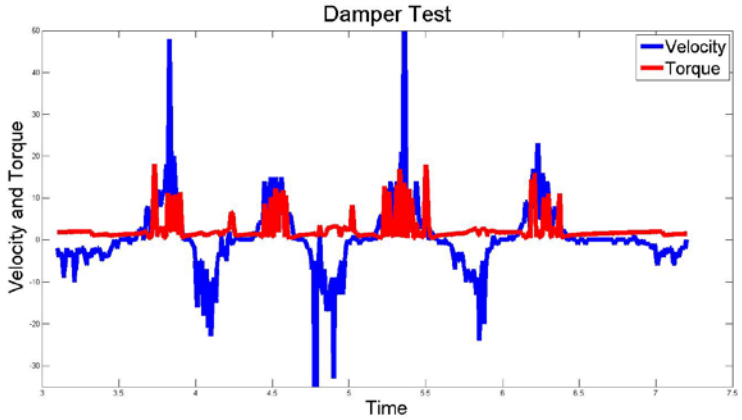
For implementing the different environments, we made use of our feedback function that can control torque. For the spring, we just set torque directly proportional to positional displacement. For the damper, we set torque proportional to velocity (calculated by looking at delta position). For the wall, we set torque directly proportional to positional displacement (like the spring), but only past a certain point, and with a very large gain. Perhaps a better implementation would use a PID loop that controls position, but this loop would need to be one sided only, and real world testing showed our method to be sufficiently effective. For texture, we set torque to resist or aid motion for seemingly random ranges of position. As these ranges get sufficiently small, the texture can feel bumpy or scratchy.

Because the microcontroller already measures or calculates torque, velocity, and position in the process of operation, we used the microcontroller as our data acquisition device. To collect data, we just had the microcontroller print all relevant data over serial a few hundred times per second. We also increased the baud rate so that collecting data wouldn't affect operation.

Below are plots of the data collected from each environment. The units for torque, velocity, and position are completely arbitrary. Torque is determined by measuring current through the motor by measuring amplified voltage drop across a current sense resistor with a 10 bit A/D. Position was measured using a 10 bit A/D to track the angle of a wheel with a significantly different diameter than the joystick itself (so angle in degrees doesn't even describe anything useful). Velocity was calculated by periodically evaluating the diff of position. Although proper coefficients could be found to translate these arbitrary measurements into standard units, all the scales are linear and as such all the relationships are accurately represented. The math on the microcontroller is done using our own arbitrary units. For this reason, we didn't bother with unit conversions.

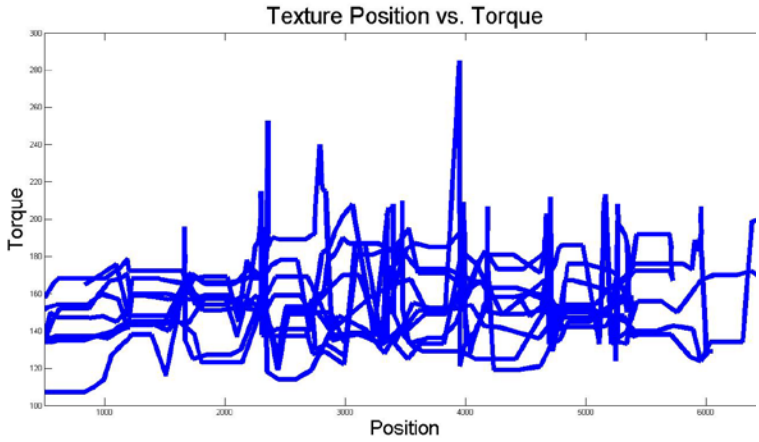


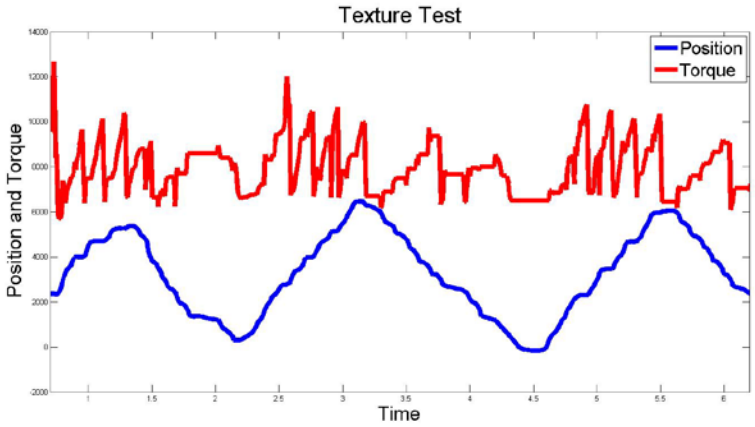
Our system has a lot of natural (mechanical) friction, and therefore damping. Nonetheless, you can see the very slight characteristic ringing from the spring.



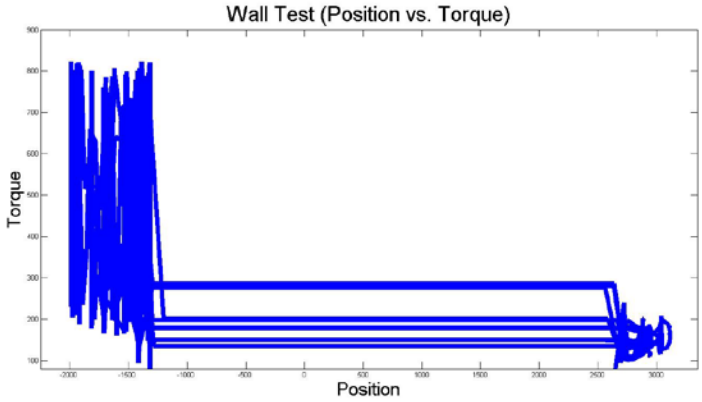
With the damper test, notice how the torque appears to only spike when velocity goes positive. This is actually an interesting artifact that may warrant a hardware change. Because the current sense resistor is on one side of the motor, when PWM is supplied on that side the PWM shows up across the resistor, giving a very noisy current signal with lots of spikes. When PWM is supplied from the other side, the voltage gets filtered across the LR filter of the motor windings, and the current signal is much cleaner. If the Damper graph was zoomed in on the negative velocity spikes, you would still be able to see a positive torque bump to counter the velocity- it's just a much cleaner signal. This could be improved in hardware by adding filtering to the current sense resistor measurement, or by placing a current sense

resistor on each side of the motor so you can sample whichever is not directly next to the PWM signal. Note that this artifact showed up in all our tests, and actually provides slightly non-symmetric physical response (since the feedback loop has a hard time finding a lock with the noisy signal), but it is most obvious in the damper graph (many of the other graphs only show one directional torque applied by the motor).





As expected, torque vs. position looks like noise, and position vs. time shows 'rough' motion.



Torque spikes massively as soon as position reaches 'the wall,' stopping the joystick from moving any further.

Video link demonstrating command line interface and physical interaction with the different environments: http://youtu.be/dGLiOeVB6Sw

Note that a copy of this video can also be found in our github.

Github link: https://github.com/carilley/elecanisms

Code for this project can be found specifically in the hapticPaddle and MP2_report folders. The main code is in virtualenvironments.c (for the microcontroller side of things) and setEnvironment.py (for the command line USB interface)