POLYTECH
Peter the Great
St.Petersburg Polytechnic
University

Course

« Introduction to Biomedical Engineering»

Dr. Kirill Aristovich

Section 4: Basics of High-level programming: Matlab
Lecture 4.3: Closed-loop control of bionic prosthetics

Closed-loop control of bionic prosthetics

Hello! And welcome to the final lecture, where we are going to design a human-robot interface in the form of robotic prosthetics.

We should start with the problem outline. So we have a robot, and we wish to control it seamlessly using smooth human interaction. Of course, we can use joystick or keyboard interface, but our task is to design something which would work intuitively and IN ADDITION to other tasks the human can perform in the meantime. Bionic prosthetics is the good example of such device, and acts as a demonstration for the platform of tech possibilities that you can employ to instrument such systems.

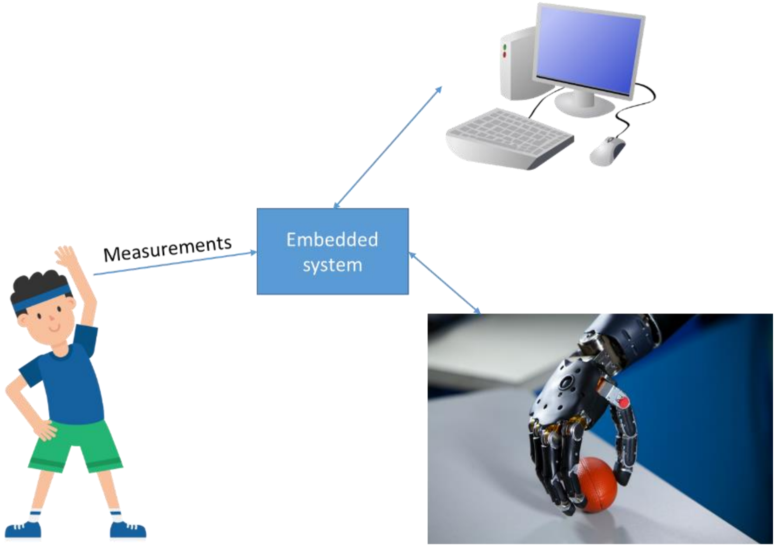


Figure 1 - Problem statement

Remembering the general structure of embedded system, we will split it to sensing part, controlled part, and a computer component. The controlled part is easy: all robotic systems are an assemblies of motors and actuators of different sort. We will simplify the system to control a single finger, which in turn can be represented as a motor, which is normally employed to move that finger. In fact, we could generalize this with the generic transfer function, but motor is more illustrative in this case.

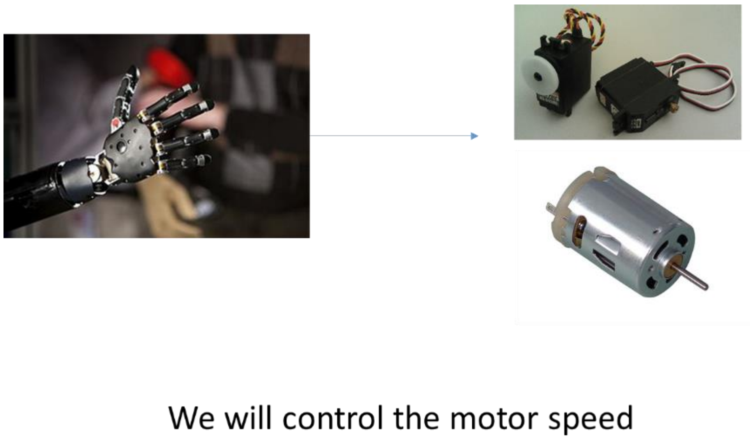


Figure 2 - Thing to control

The sensing part can be also generalized as a signal in time, however we are going to be using the EMG. Remember from the first section, that this is a voltage difference signal, which we can sense from the surface of, say, hand, using the electrodes. The physiological origin of the signal is in so-called muscle action potentials, which occur when muscle fiber contracts, and in fact IS a direct cause of this contraction. With more frequent activity the muscle fibre contracts stronger. Also, with more fibers involved, we start to get more of these action potentials from different places. In the end, from a sensor perspective, the amount of chaotic activity in EMG is generally proportional to the force of muscle contraction.

- EMG (Electromyogram) is the surface potential measurements of the muscle activity:
  - Each muscle fibre generates action potential when active
  - More fibres → More activity → stronger force
  - More fibres → More activity on the EMG

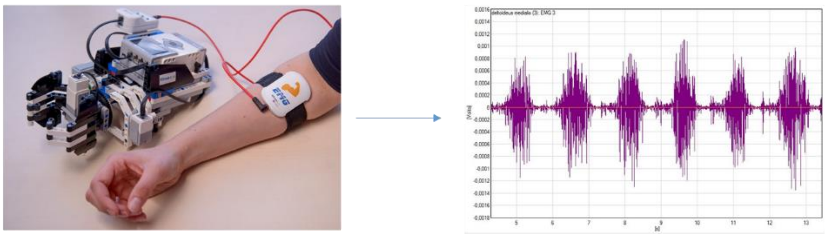


Figure 3 - Thing to measure

The overall characteristics of the signal are following: It is stochastic, Has a frequency band of 1 to a 100 Hertz, and it needs amplification as the order of magnitude is millivolts. These potentials are stronger if they are closer to the electrode, so the singe electrode contains most information about the NEARBY muscle. We can use several electrodes located above different muscles to resolve complex motion, or gestures where several muscles are active.

- Stochastic
- Frequency band 1-100 Hz
- Needs amplification
- Potentials are stronger if they are closer to the electrode → Location of the electrode gives information on the nearby muscle → several electrodes to resolve complex motion where several muscle are active

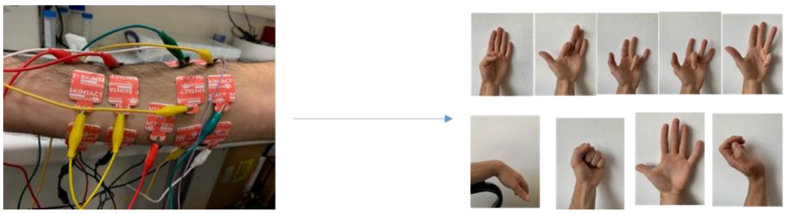


Figure 4 - Characteristics of the signal

So again, without loss of generality we simplify the system down to one motor and one EMG sensor, knowing that we can add more sensors and motors to the system if we need to.

Let's draw a diagram of this, noting the signals flow: From human – EMG needs to be amplified, filtered (which we already know how to), and sent to the controller. Controller needs to control the motor real-time, and by mere coincidence, we already know how to do exactly this from the last lecture of the third section.

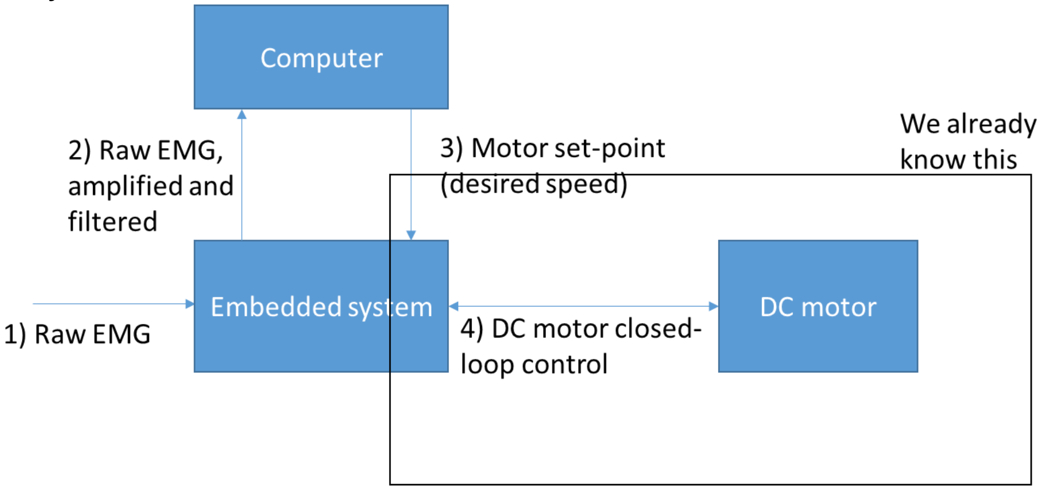


Figure 5 - For now, lets concentrate on controlling one finger with one sensor

The central ‘brain’ of the system is a computer, which communicates with the microcontroller by reading the EMG signal, and makes the decision by generating the demand, or desired motor speed. Once we have established the system operation, this architecture ensures reliability and real-time performance for the key components which we do not need to modify, and great flexibility in experimenting and tuning by just using the high-level software.

Here is the list of components we would need: First, we need all the stuff for the motor control from section 3. In addition we will use Olimex Arduino EMG shield, and sticky EMG/ECG standard electrode. The Olimex is a shield with built-in amplifier and filter, and the sticky electrodes are basically pads with conductive gel that you can connect to a wire. They are standard and available even on Amazon.

This is how the full assembly should look like. You would use 3 electrodes: 1 ground, and 2 for connecting to the Amplifier, which, if you remember, amplifies voltage difference between them. You would position one over the muscle, and the other on the skin away from any muscle to act as a stable reference.

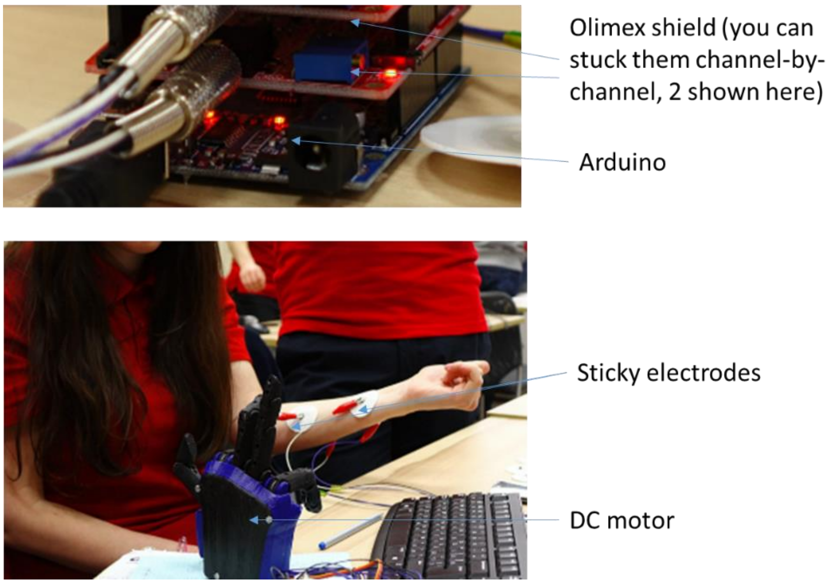


Figure 6 - Full assembly

Now, we can start with the controller setup. We need to read the signal from the shield, so we would want to understand which pin to read, and follow the instructions form shield manufacturer for specific clock settings, then we need to organize the data structure, or…

That is 17!

- #define NUMCHANNELS 6
- #define HEADERLEN 4
- #define PACKETLEN (NUMCHANNELS * 2 + HEADERLEN + 1)

- #define SAMPFREQ 256   // ADC sampling rate 256
- #define TIMER2VAL (1024/(SAMPFREQ))   // Set 256Hz sampling frequency

Figure 7 - Data structure and transmission

We can ignore all of it and open the example included in the Olimex package. Let's go over the pre-processor and setup. The file sets up some basic data structure already, and uses 17 bytes in data packet, which is basically an array. First 4 bytes are used for the header (we can safely ignore them for now). Then there are 12 bytes for 6 channels. Each channel will use 2 bytes (16 bit resolution per channel), and yes, you can connect up to 6 of Olimex shields stacked on top of each other in order to have 6 channels. This is a good example of code which accounts for future expansions.

- volatile unsigned char TXBuf[PACKETLEN]; //The transmission packet
- volatile unsigned char TXIndex;   //Next byte to write in the transmission packet.
- volatile unsigned char CurrentCh;   //Current channel being sampled.
- volatile unsigned char counter = 0;   //Additional divider used to generate CAL_SIG
- volatile unsigned int ADC_Value = 0; //ADC current value

Figure 8 - Since we are using interruptions

The code uses timer interruptions to establish the reliable sampling rate of 256 per second. Here is the declaration of timer 2 overflow interruption which occurs every 1/256 th of a second and runs the 'Timer2_Overflow_ISR'. The last bit of setup code established serial communication.

```
void setup() {

 noInterrupts();  // Disable all interrupts before initialization

 // LED1
 pinMode(LED1, OUTPUT);  //Setup LED1 direction
 digitalWrite(LED1,LOW); //Setup LED1 state
 pinMode(CAL_SIG, OUTPUT);

 //Write packet header and footer
 TXBuf[0] = 0xa5;    //Sync 0
 TXBuf[1] = 0x5a;    //Sync 1
 TXBuf[2] = 2;       //Protocol version
 TXBuf[3] = 0;       //Packet counter
 TXBuf[4] = 0x02;    //CH1 High Byte
 TXBuf[5] = 0x00;    //CH1 Low Byte
 TXBuf[6] = 0x02;    //CH2 High Byte
 TXBuf[7] = 0x00;    //CH2 Low Byte
 TXBuf[8] = 0x02;    //CH3 High Byte
 TXBuf[9] = 0x00;    //CH3 Low Byte
 TXBuf[10] = 0x02;   //CH4 High Byte
 TXBuf[11] = 0x00;   //CH4 Low Byte
 TXBuf[12] = 0x02;   //CH5 High Byte
 TXBuf[13] = 0x00;   //CH5 Low Byte
 TXBuf[14] = 0x02;   //CH6 High Byte
 TXBuf[15] = 0x00;   //CH6 Low Byte
 TXBuf[2 * NUMCHANNELS + HEADERLEN] =  0x01;  // Switches state

 // Timer2
 // Timer2 is used to setup the analag channels sampling frequency and packet update.
 // Whenever interrupt occures, the current read packet is sent to the PC
 // In addition the CAL_SIG is generated as well, so Timer1 is not required in this case!
 FlexiTimer2::set(TIMER2VAL, Timer2_Overflow_ISR);
 FlexiTimer2::start();

 // Serial Port
 Serial.begin(57600);
 //Set speed to 57600 bps

 // MCU sleep mode = idle.
 //outb(MCUCR,(inp(MCUCR) | (1<<SE)) & (~(1<<SM0) | ~(1<<SM1) | ~(1<<SM2)));

 interrupts();  // Enable all interrupts after initialization has been completed
}
```

Malarkey

Data structure, note from this:

- 6 channels
- 2 bytes per channel
- Position of the bytes are important, especially when you are receiving all this stuff

Set up timer interruption with required frequency → Now the function Timer2_Overflow_ISR will trigger 256 times a second

Sending the data to a computer via serial interface (remember the settings!)

Figure 9 - Setup

There is basically nothing in the main loop, all the stuff is happening in the ISR. Let's go over it. Ignoring the malarkey of switching LED, there is a loop which goes through the channels 1 to 6, and reads the associated analogue pin. Then the value gets converted from integer to 2 bytes, and stored in the array in the appropriate positions for each channel. After the loop is done, we are sending the array to the serial, one by one. Ignoring another maintenance malarkey which is not important, that is basically it!

```
/****************************************************/
/*  Function name: Timer2_Overflow_ISR              */
/*  Parameters                                      */
/*    Input   :  No                                 */
/*    Output  :  No                                 */
/*    Action: Determines ADC sampling frequency.    */
/****************************************************/
void Timer2_Overflow_ISR()
{
  // Toggle LED1 with ADC sampling frequency /2
  Toggle_LED1();

  //Read the 6 ADC inputs and store current values in Packet
  for(CurrentCh=0;CurrentCh<6;CurrentCh++){
    ADC_Value = analogRead(CurrentCh);
    TXBuf[((2*CurrentCh) + HEADERLEN)] = ((unsigned char)((ADC_Value & 0xFF00) >> 8));  // Write High Byte
    TXBuf[((2*CurrentCh) + HEADERLEN + 1)] = ((unsigned char)(ADC_Value & 0x00FF)); // Write Low Byte
  }

  // Send Packet
  for(TXIndex=0;TXIndex<17;TXIndex++){
    Serial.write(TXBuf[TXIndex]);
  }

  // Increment the packet counter
  TXBuf[3]++;

  // Generate the CAL_SIGnal
  counter++;    // increment the devider counter
  if(counter == 12){  // 250/12/2 = 10.4Hz ->Toggle frequency
    counter = 0;
    toggle_GAL_SIG(); // Generate CAL signal with frequ ~10Hz
  }
}
```

Malarkey

For each channel

Read the value

Convert to bytes

Send the structure over serial

Some maintenance/malarkey stuff to be in sync

Figure 10 - What happens every interruption?

We have organized the seamless 256-a-second digital data transmission. As an exercise question to check yourself, what do you think is Olimex antialiasing filter cut-off frequency?

Right, next bit to organize is the computer side of things to read the data and make the decision on motor speed, sending it back to controller. We will obviously use Matlab, so let's briefly go over the setup and actual operation.

```
ard= serial('COM1','BaudRate',57600);

Fs=256; % set on ard code

Twindow =1; % number of seconds to have on screen at once
plotsize=Twindow*Fs;
chn_num=2;

time=(0:plotsize-1)/Fs;
data=zeros(chn_num,plotsize);

packetsize=17;
numread=20; %max 30 as 512 bytes inbuffer

%% Graph Stuff
plotGraph1 = plot(time,data(1,:),'-',...
    'LineWidth',2,...
    'MarkerFaceColor','w',...
    'MarkerSize',2);

title('EMG','FontSize',20);
xlabel('Time, seconds','FontSize',15);
ylabel('Voltage, V','FontSize',15);

drawnow
```

Open serial (baud should match with arduino)

Remember sampling frequency, we need it to compute time

Data array, where you store all data, we will use cycling buffer only storing last second (in this case 256 values), make everything zero in the beginning. 2 channels because why not?

This is the structure size, need to match what you send, right?

We will set up the graph now, only updating its data in future

Figure 11 - MLAB side of things: setting up

First we open the serial, remembering to check baud rate to be the same as in Arduino. Next, we are setting some parameters we are going to need for plotting (we want to plot 1 second, but we also want to make it flexible), and the data structure we are going to use: here we have selected just two channels for now, each will have 256 values (or 1 second) stored, and we will use this as a circular buffer: when the array is filled, we are going to start recording over it again, like a clock.

```
%% Reading Data

iSample =1;

while ishandle(plotGraph1)

    if ard.BytesAvailable >= numread*packetsize

        for iRead = 1:numread

            [A,count] = fread(ard,packetsize,'uint8');

            data(1,iSample)=double(swapbytes(typecast(uint8(A(5:6)), 'uint16')));

            iSample = iSample +1;
            if iSample > plotsize
                iSample =1;
            end

        end


        try
          set(plotGraph1,'YData',data(1,:));
        catch
        end

        drawnow

        packetsleft=floor(ard.BytesAvailable/packetsize);

        if packetsleft > 20
            fprintf(2,'Update rate is too slow!: %d \n',packetsleft);
        end

    end

end

fclose(ard);
```

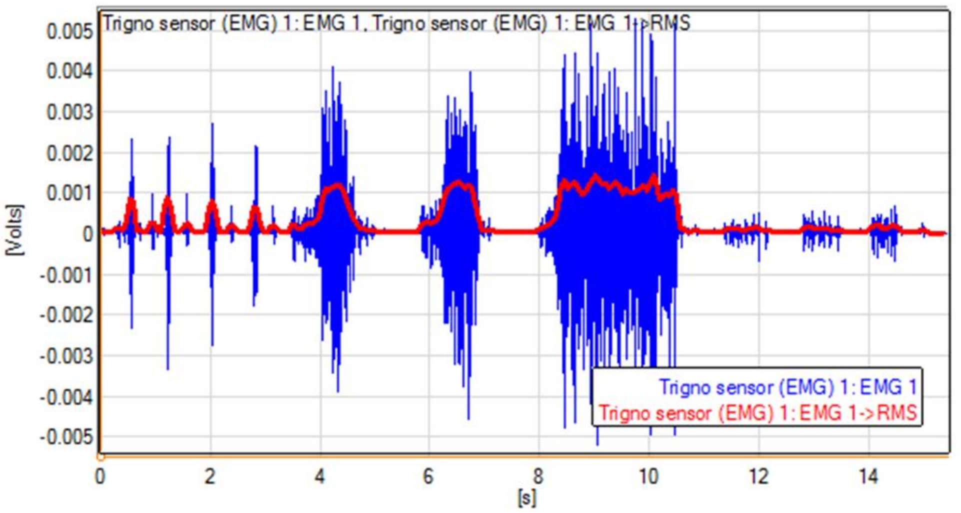
- iSample is a counter
- Do forever (if the graph is still there)
- If there is enough data to read in the buffer (we will read chunks of 20, cause why not? It is trade-off between speed and processing)
- Read the data into big data matrix
- Convert data from 2-byte to double prescision
- Increment counter, if more than 1 sec, start over again (circular buffer)
- Update the graph (there is try-catch to google)
- This is to check that you can process faster than you read (chunks of 20 works on my computer)
- This would occur if you close the graph window

Figure 12 - Matlab side of things: reading data

Now, we will set a graph, where we plot the data. The strategy here is to pre-set everything like window and plot parameters, and just update the data itself so save time.

Ok, we go into actual operation loop, where we set the sample number to 1, and start the loop while the window is there – this way upon closing the window the loop terminates automatically. We check if there is something in the serial buffer, and if there is enough information for us to read, we read the specified number of samples, which in this case is 20. (It gave a good balance for my laptop between the speed of reading the data and speed of doing everything else). Now, for each sample we read the structure from the buffer, and assign the first channel of EMG to our first channel of data, converting it from bytes to double precision numbers. Then we increment the counter, remembering that if it reaches the end of the array (256, or 1 second), it should start from 1 again, overwriting the first values as circular buffer would.

$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$



Measure of the spread (or power of the spread, If mean =0, RMS = STDEV)

Figure 13 - RMS computation

Then we update the graph, after which we make sure that our buffer does not overflow by growing too fast. In the very end we note a statement that closes the serial in case we have closed the window.

FEW, that's it! Now when we start the Arduino, and start our Matlab script, we will see the EMG in real time on the screen! The only thing left to do is to make a decision on the demand – our motor speed, based on what we receive. The usual way to compute the 'spread', or 'power' of the stochastic process (proportional to muscle activity in our case) is to compute the standard deviation, which is equal Root Mean Square in case of zero-mean signal. In our case this gives powerful way to control the motor action, for example by just making its speed to be PROPORTIONAL to RMS. This makes some sense: no muscle activity will produce no motion, and more active action would produce faster motion. It is not ideal, but that's a beauty of high-level programming, we easily add more complicated stuff later.

We can leverage the power of matlab adding simple code to compute and display the RMS on a second graph. Then we can scale it: balance the maximum speed you want to the maximum RMS your muscle can produce empirically. And finally, send this back to arduino in order to set up the speed.

I trust that you can do it yourself like a no-brainer now!

In summary, we have created a closed-loop speed controlled motor (finger), operated real-time by EMG muscle activity! Now we have this structure in place, it is clear to see how we can easily expand on this: Several fingers controlled by several sensors, or more sophisticated analysis to control the motors better.

In the very end, there are questions to ask yourself: Can you comfortably replace the motor with a linear actuator? Can you control forward-backward motion? Can you make the system to sense additional parameters (pressure when touch, temperature, etc) given appropriate sensors? Can you think of a way to analyze this data in Matlab and send back improved demand for the motor? Can you make the Matlab code which expands on these and incorporate more ideas?

If the answer to all of this is **YES**, then I have successfully achieved my goal, I wish you all the best, and Bye-Bye!

Some graphic material used in the course was taken from publicly available online resources that do not contain references to the authors and any restrictions on material reproduction.

This course was developed with the support of the "Open Polytech" educational project

OPEN POLYTECH

Online courses from the top instructors of SPbPU