Lab sessions

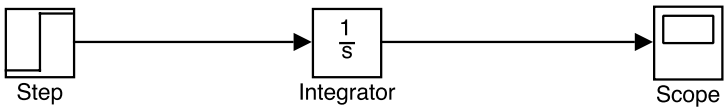
In these lab sessions we will design, construct, and test a light-sensing solar tracker: a device that detects the direction of maximum light flux and orients a solar panel accordingly.
To keep things simple we will use the Lego Mindstorms EV3 set as the hardware for our tests, with the Lego light sensor as our mock solar panel. Our objective is to design a 1-degree of freedom tracker, which can move the solar panel around the vertical axis towards the direction of maximum solar flux. We will use a closed-loop control algorithm to position the panel, and a continuous optimization algorithm to compute the optimal panel orientation from the smallest possible number of light flux readings. A successful design will have to achieve precise positioning in a reasonably short time and using as little energy as possible.

Session 1: introduction

INTRODUCTION TO SIMULINK

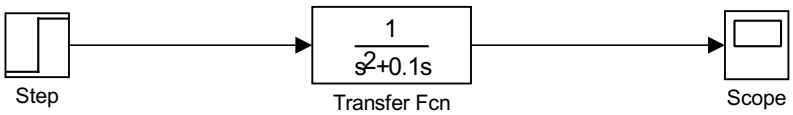
Simulink is a graphical simulation environment. It allows to design and run simulations in the form of block diagrams, for example using blocks such as the ones we use to represent linear systems (integrators, gains, etc.).

1)
To warm up, open the latest release of Matlab, and open the Simulink Library.
Select file->new->model, and draw the following model:



By default the step block outputs a step beginning at time t=1 with amplitude 1, hence the output plotted by the scope should stay at 0 for t<1 and increase with slope 1 for t>1. Take some time to explore the simulation options, change the simulation time, change the step block parameters, rescale the scope axes.
Save the model on the Desktop as "Lab1_1".

2)
Let's change the integrator with something more fun. Delete the integrator block, select "Transfer Fcn" from the Continuous set of the Simulink library, and plug it in place of the integrator. Edit the block so transfer function is $\frac{1}{s^2+0.1s}$ (set the numerator to [1] and the denominator to[1 0.1 0] in matlab jargon), and change the simulation time to 50 (to leave sufficient time for the transient dynamics to die off).

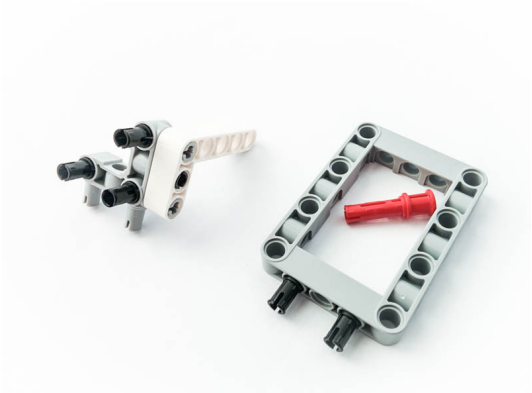


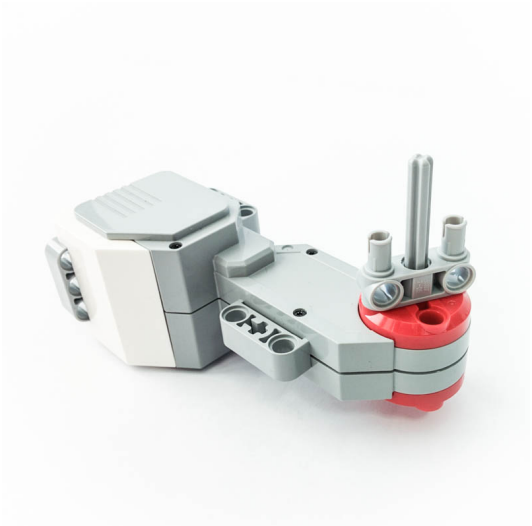
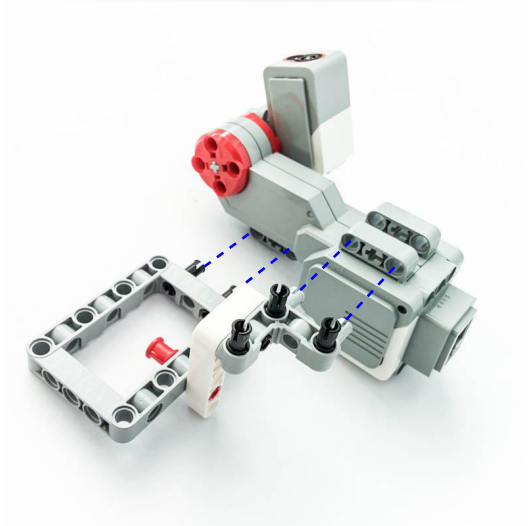
The Laplace transform of the output, given a step input, is $\frac{1}{s^3+0.1s^2} = \frac{-100}{s} + \frac{10}{s^2} + \frac{100}{s+0.1}$, which is equal to $-100 + 10t + 100e^{-0.1t}$ in the time domain (check!). Verify that this is the case in the scope plot.

Now repeat the experiment after putting an integrator (1/s) before the transfer function block. Compute the output first by hand, then check against the scope plot. Try moving the integrator after the transfer function, does the output change? Why? Save the model on the Desktop as " Lab1_2".

3)
Now we'll learn to interface Simulink with the EV3 brick, and to control an electric motor.

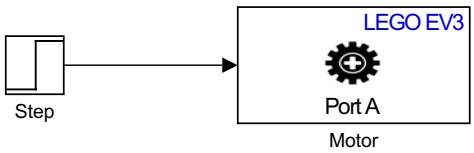
First, build the Lego system as in the figure

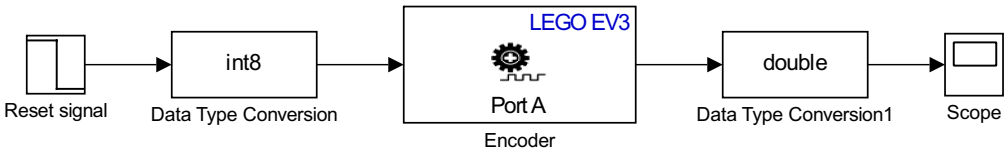


Let's start with setting up a connection to the EV3 brick:

- plug the motor data cable into port A of the Lego brick
- build a model as in the following figure.





  - Double click on both motor and encoder and select "EV3 brick output port: A".
  - In the Encoder block, select "Reset mode: Reset by external signal".
  - The reset signal is a standard Step block, double click on it and select Step time: 0.1, Initial value: 1, Final value:0. This will reset the encoder output to 0 during the first 0.1 seconds, then let it free to record the motor position.
  - Edit the step input with final value 20 (20% of the maximum motor torque), and leave the step time at 1, so that the input is given well after the reset signal has turned off.
  - The two data type conversion blocks are used to translate the double type signals output by the step and read by the scope into int8 (8 bit integer) signals handled by the encoder. Drag them into place and select the correct data type by double clicking on the block and editing the field "Output data type".
- select simulation -> mode -> external
- select tools -> run on target hardware -> prepare to run
- in the window "configuration parameters", section "run on target hardware", select target hardware -> LEGO Mindstorms ev3, then set the ip address reported by the brick (you can find it in the Brick Info menu)

For the moment remove the data cable from the sensor, so that the sensor block is free to rotate. Press play and check if it works. Remember, the motor takes inputs between -100 and +100, signals beyond this interval are clipped. Save this model in the Desktop as " Lab1_3".

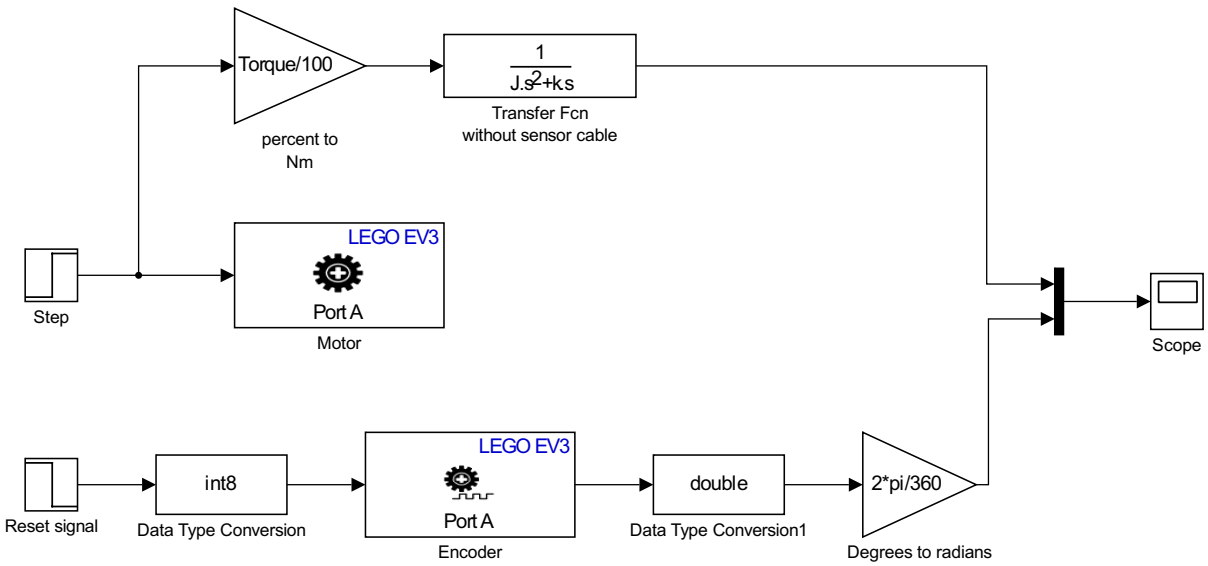
4)
Let us try to build a model of the actuator and identify its parameters through some trial and error. Start with some physical reasoning: a very coarse model of the motor is given using Newton's law, and assuming that the only forces acting on the axis are the motor's torque, and a viscous friction. This gives the model

$$J\ddot{x} = u - k\dot{x}$$

where x is the angular position (in radians), J the moment of inertia (in kg m^2), u the applied torque (in Nm), and k a frictional torque coefficient. The corresponding transfer function is

$$X = \frac{1}{Js^2 + ks}U$$

This is similar to our Lab1_2, so we can run the model alongside the real motor and try to match the model's parameters to those of the real system. Copy the transfer function from Lab1_2, and connect the diagram as in the following figure.



The triangles are Gain blocks (from the Math Operations block set). The gain block "percent to Nm" is used to translate the input provided by the controller (which is measured in percent of the total available torque) into torque, so that the quantities of the transfer function following the block can be expressed in terms of the physical properties of the motor and sensor.

Now, by trial and error, try to find your best match of the system's parameters. To guide your search you can use the following data:

- the maximum torque of the motor (obtained when the input signal is 100) is equal to 0.2Nm.
- the weight of the color sensor is approximately 15g, to which you must add the mass of the moving parts of the motor. To convert this into the moment of inertia remember that the moment of inertia of a body is roughly equal to MR^2, where M

is the mass and R the radius (assuming the body is roughly spherical with rotation axis going through its centre).

- by the final value theorem (which you will see during class) the slope of the step response of our transfer function is equal to $\frac{U}{k}$, where U is the motor torque in Nm and k is the frictional torque.

Note that the simulation step size in the above diagram is set by the sample time of the Encoder block, and might be too large to integrate correctly certain values of the parameters. If the simulations don't match what you expect from the transfer function you can change the solver algorithm by editing Simulation->Model Configuration Parameters->Solver. Ode14x seems to be rather robust.
You can save this model as Lab1_4, remember to save it on a device of yours as this computer's disk might be cleaned. Take note of the parameters, as they will be needed in the next lab session.

5)
(Optional)
It is possible to identify the model through automatic methods (like least square fitting). Try to do it using the command

>>tfest(data, number of poles, number of zeros)

which is found in the System Identification Toolbox.
Hint:

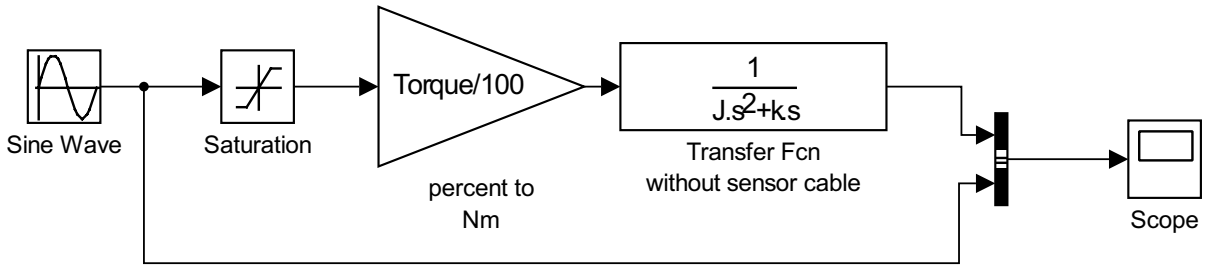
1. to obtain input and output data from the model you can use two "To Workspace" block.
2. Double-click on the block and select Save Format: Structure with Time, as the default selection (Timeseries) does not work with models run in External mode.
3. Extract a vector from the structure using the command

   >> output = simout.signals.values
4. prepare the argument of tfest through the command

   >>data=iddata(output,input,sampling time).

Session 2: actuators

Today we will design a control law for the motor.
1)
Draw the following model, using the parameters identified during Lab Session 1 for the motor transfer function.



For a more realistic setup we have included a saturation block after the sine input. Double click this block and set the upper limit to +100 and the lower limit to -100 to match the behaviour of the motor controller. For the moment, leave the simulation in normal mode (i.e. not external), as we will play with this simulated model for a while. We have added a Bus Creator block (in the Signal Routing block set) to merge the input and output signals before the scope, so that we can plot the two signals in the same figure.

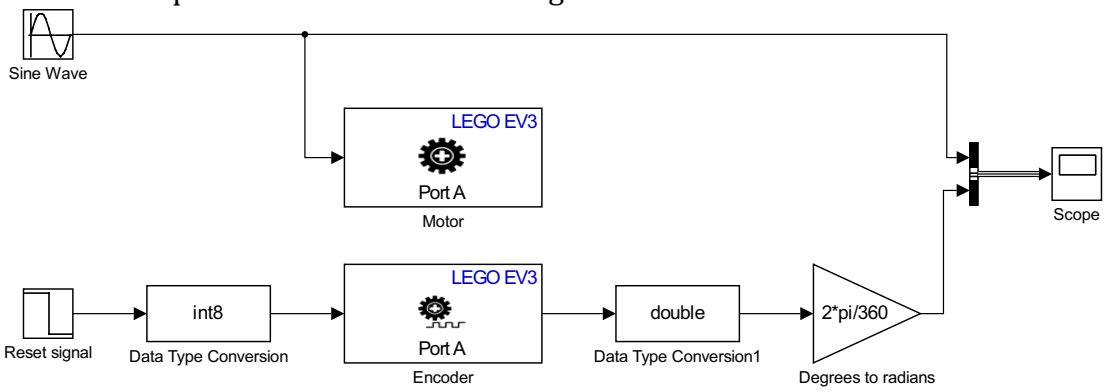
- Run the simulation with a sine amplitude of size 20 (20% of the motor maximum torque), you will observe a phase as well as an amplitude difference between the input and output signals. Try changing the frequency of the input signal (but leave the amplitude fixed). How do the phase and amplitude change?

Save the model as Lab2_1.

2)
The changes in the output amplitude observed before can be explained using the *Frequency Response Theorem*: for a given transfer function $F(s)$ and for a sinusoidal input $A\sin(wt)$ the output is equal to

$$|F(iw)|\, A\sin(\, wt\, +\, \measuredangle(F(iw))\,)$$

Let's check how good is our model by testing the real motor with sinusoidal inputs of different frequencies. Build the following model

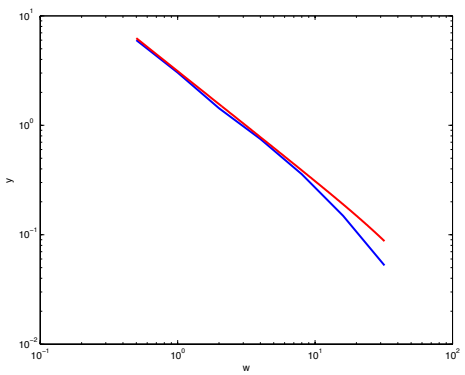


and save it as Lab2_2.
The reset signal is a step block with step time 0.1, initial value 1, and final value 0, and is used to reset the encoder to 0 at the beginning of the experiment.

- Set the sine input amplitude to 20, and the (angular) frequency to 0.5, and run the model in external mode (i.e., on the Lego brick).
- Record the angular frequency in the variable w
  >>w=0.5
- Measure the amplitude of the output signal (remember: amplitude = (max-min)/2), move to the main Matlab window and record the amplitude in the variable y
  >> y =
- Now repeat the experiment with w = 1, read the amplitude of the output and add the angular frequency and amplitude to the variables w and y by typing
  >> w=[w 1]
  >> y=[y (Amplitude you have measured)]
  Repeat for w = 2, 4, 8, 16, 32.
  Notice that this experiment requires reading relatively high-frequency signals (w = 32 is a sinusoid at roughly 5Hz), and in order to have a readable output we need to sample at a much higher frequency (twice as big by Shannon's theorem, much more if we are lazy and don't want to do any fancy signal analysis). To avoid all problems change the encoder sampling time to 0.01, much higher frequencies are not supported by the hardware.

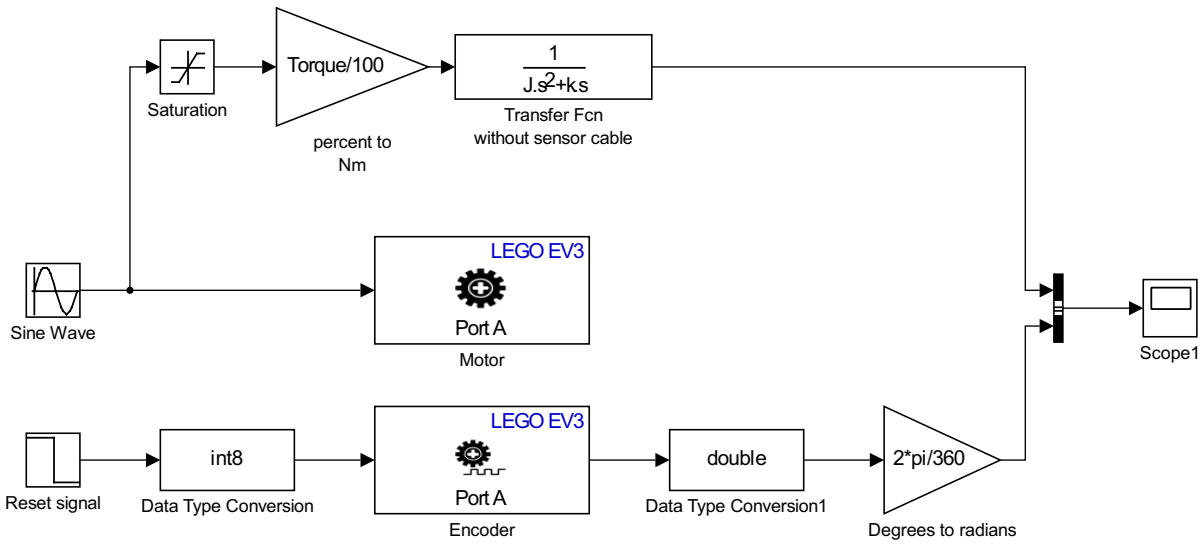
  During the above experiment you may observe a drift of the output mean value, which is not predicted by the linear model. Can you explain this drift?

- Type
  >> plot(w,y)
  to plot the frequency response of the real system. Now compute the frequency response of our linear model using the frequency response theorem:
  >> ysim=abs(20*Torque/100* (1./((J*w*i).^2 +k*w*i)))
  where Torque, J and k are your model's parameters (i.e. the parameters of the blocks in the diagram of Lab2_1). Plot the frequency response of the linear system in red on top of the previous one typing
  >>hold on
  >>plot(w,ysim)
  and change the axes to logarithmic scaling in Edit->Axes Properties. You should obtain something like this:



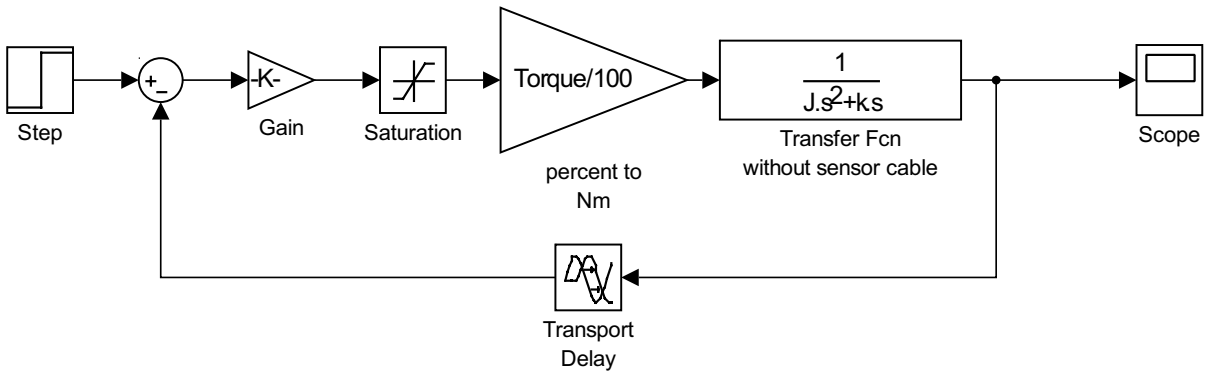
which shows a pretty good match. Notice that the error increases at high frequency, since the measurement errors and unmodeled dynamics become more prominent in this range.

3)
Let's test the model when we step outside of the (quasi-) linear range. Put the two diagrams we built before side by side in the same model as follows.



Save as Lab2_3. Run the model with a sine wave of amplitude 20, and then with one of amplitude 150, and check if they behave similarly (they should).

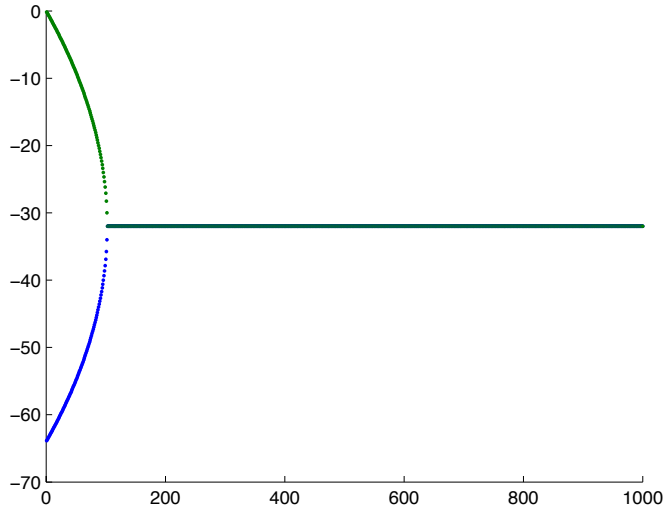
4)
In order to construct our solar tracker we need to design a control law to precisely place the motor at a given angular position. Draw the following model and save it as Lab2_4.



Ignoring the saturation, and calling F(s) the transfer function of the open loop system, the transfer function of the feedback system pictured above is $\frac{F(s)}{1+F(s)} = \frac{\text{Gain}\cdot\text{Torque}}{100Js^2+100ks+\text{Gain}\cdot\text{Torque}}$.

The negative feedback tries to keep the value of the output equal to that of the input, and the stronger the gain the stronger the effect of the feedback. However, since a feedback connection does not preserve the stability of the open loop system, we must make sure that we do not make our system unstable by choosing too large a gain.
Plot the poles of the above transfer function (the roots of the denominator) for gain values between 1 and 1000 by typing

```
>>hold on
>>gain = 1:1000;
>> for i=1:1000
plot(gain(i),real(roots([100*J 100*k gain(i)*Torque])))
end
```

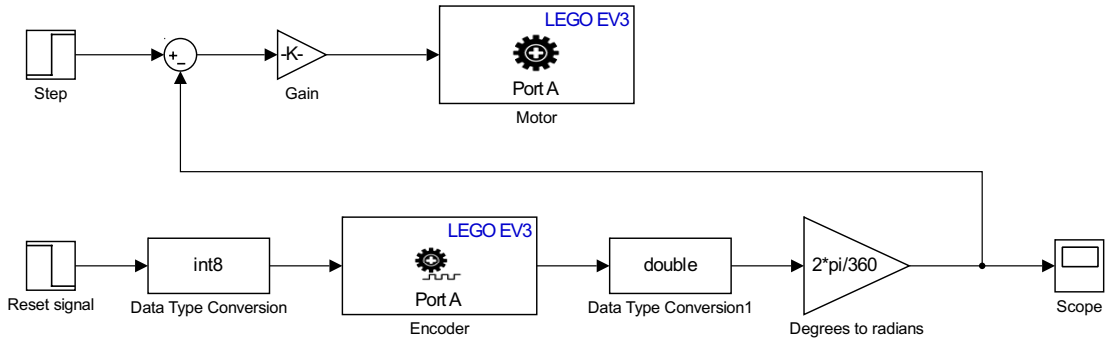


The closed-loop system has two real and negative poles for small gain values, which become complex conjugate with negative real part for larger gain values. Hence we expect a system that is always stable, but may oscillate towards its equilibrium if the gain is too large.

Is this result robust to modelling errors? Assume for instance that the output is measured with a little delay (for example 0.01 seconds), due to signal propagation and the discrete nature of the digital control. Is the feedback system stable in the presence of such a delay? We can test this by adding a Transport Delay block (from the Continuous block set) on the feedback line, and setting the Time Delay to 0.01. Simulate Lab2_4 with the delay block and test what happens for different gains. Choose a gain value that brings the system quickly to the equilibrium without too many oscillations.

Note: robustness to delay can be more properly discussed using gain and phase margin, if you can take some time to look these concepts up, and try to apply them in the case of our model.

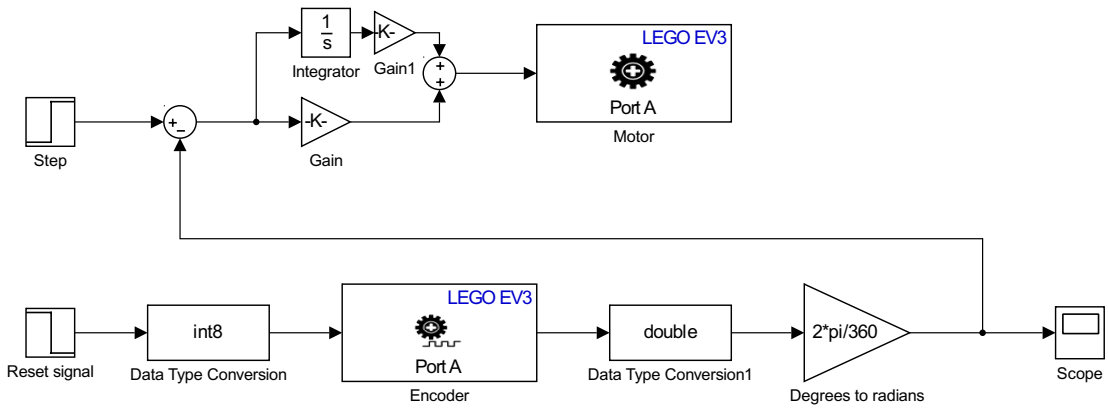
5)
Now we will try to implement the feedback law we designed before on the real system. Draw the following diagram and save it as Lab2_5, set a step size equal to pi, and set the gain equal to the value chosen at the previous step.



If the gain is not too big the motor should complete about half a turn, but you will probably notice a slightly different response than the linear model, for example more oscillations and some steady state error (i.e. the motor does not stop exactly at pi). Also, notice that now if we increase the gain too much the system becomes unstable (probably because of the delay?). Try this carefully, and remember to set a simulation time not too big (10 seconds should be more than enough) so as not to stress too much the motor components. Set the gain back to the value chosen at step 4.

Now plug the data cable into the color sensor, and connect it to port 1. The cable acts as a nonlinear spring on the system, adding one more unmodelled effect. Run the system again and see how it behaves. These differences in dynamics are due to small (or not so small) errors in our model, and the steady state error in particular is due to a rather large amount of friction and slack in the motor gears. This can be a problem in our application, as we need to place the color sensor within a few degrees of the desired position.

6)
To improve the performance of the control law we can put an integrator in parallel with the proportional gain, realising a PI controller (Proportional Integral).
Draw the following diagram and save it as Lab2_6.



Change the integral gain until you find a good compromise between speed and amplitude of the oscillations. You may also try small changes in the proportional gain to see if you can obtain a better positioning. Try to find P and I gains so that the sensor reaches its desired position, within a couple of degrees, in at most ten seconds. We could design an even faster controller, but since our system is expected to complete a set of measurements only a few times every hour, this is a reasonable target for the

moment. (The choice of the PI gains can be somewhat complex, there are a number of guidelines to direct this choice, see e.g. the Ziegler-Nichols method.)

Session 3: sensors

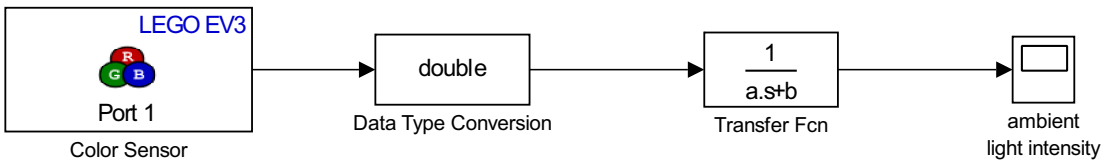
Let us start by checking the performance of the color sensor, and designing a filter to reject disturbances such as objects rapidly passing in front of the sensor.

1)
Start with a blank model, and place the Color Sensor and Scope blocks as in the figure.

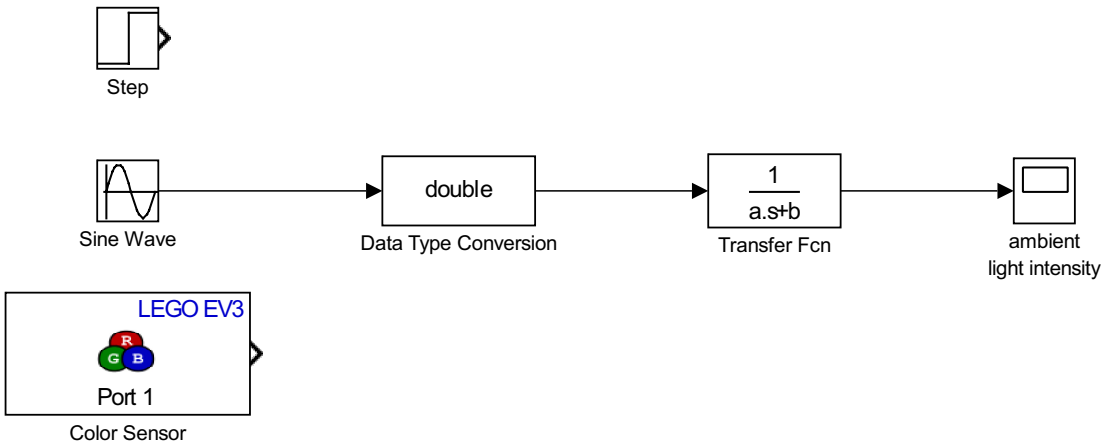
Double click colour sensor block, select mode:ambient light intensity.
Save the model as Lab3_1.

Plug the sensor into port 1 of the brick, run the model in external mode, and wave the sensor in front of a source of light. You will see that the signal changes rapidly, meaning that the dynamics of the sensor is rather fast.

2)
For our system we may need to filter out small disturbances, such as somebody walking in front of the sensor. We can do this by letting the sensor's output through a linear filter, as in the following figure.



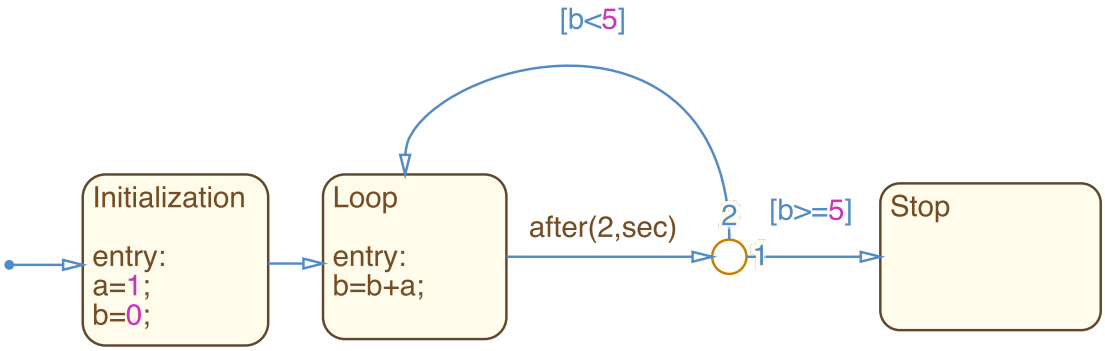
Save the system as Lab3_2. Try the effect of different transfer functions on the filtered signal in order to obtain a system whose step response converges within 1% of the equilibrium in less than 10 seconds, while reducing the amplitude of a sine-wave disturbance at frequency 3.14rad/sec (1/2Hz) by at least a factor 10. You can test the performance of the filter first in simulation by substituting the color sensor with a computer-generated signal:



Hint: the filter can be designed as the series connection of two first-order filters of the form 1/(as+b). In the Matlab command line you can use
>>N=[1]
>>D=[a b]
>>sys=tf(N,D)
>>step(sys,10)
to see a plot of the step response of the system defined by vectors N and D for times between 0 and 10, while you can use the frequency response theorem (amplitude = |F(iw)|) to compute the amplitude of the output given an input at frequency w (if you have seen it in class, you could get the frequency-amplitude response from the Bode diagram of the transfer function).
Hint : given two polynomials (as+b) and (cs+d) described by the vectors [a b] and [c d], you can compute the coefficients of the product of the polynomials as >> CoefficientsOfTheProduct = conv([a b],[c d])

3)
We need to design a finite state automaton that collects a number of evenly spaced sensor readings, which will be used to decide the angle of maximum brightness. We will use the library Stateflow in Simulink.

A finite state automaton is a graphical representation of a program in the form of a diagram. The advantage of using a Stateflow automaton instead of a standard Matlab function is that the automaton can easily incorporate temporal specification (such as assign a particular value to a given variable after n seconds), which are useful to control our system in time. In Stateflow, an automaton is represented by a set of states, each of which contain assignment instructions like the ones you can type in the Matlab command line (e.g. a=1, b=[1 2 3]), and transition condition, which define the conditions under which a transition between two states happens. For starters, design the following diagram using the elements in the Stateflow library.

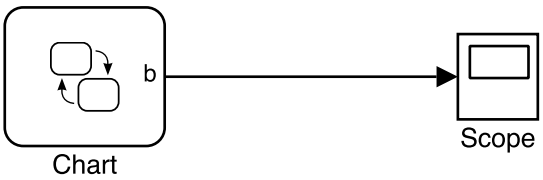


Click Chart-> Add Other Elements -> Local Data, and add the local variable a. This variable is visible to all blocks of this chart (which behaves as a single Matlab function), but is not visible outside the chart. Then click Chart -> Add Inputs & Outputs, and add the output b. This makes b available to the simulink environment: go back to the main simulink environment by clicking the upward arrow and you will see that the chart block now has an output named b.

Let's see what the different ingredients of the above chart mean:

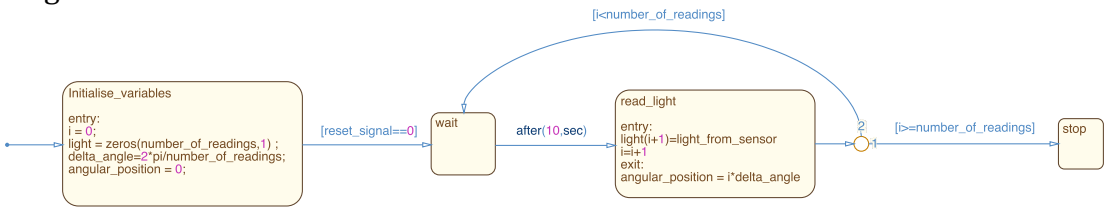
- The arrow on the left is the default transition, which is executed at the beginning of the simulation.
- The first line of each block is simply the name of the state, it has no effect on the code.
- The keyword "entry:" means that all following instructions are executed only once each time the state is entered. Alternatively one can use "during:", meaning the instructions are repeated continuously (at each simulation time step) while in the state, and "exit:", meaning the instructions are executed once when the state is exited.
- The keyword "after(2,sec)" on the transition is an event (in this case, a clock measuring 2 seconds). In general events are written as instructions with no brackets over a transition, and the transition is executed only when the event takes place
- The keywords "[b<5]"and "[b>=5]" are conditions: the corresponding transition takes place only if the condition is verified. In general, conditions are written between square brackets.

Now go up one level to the main Simulink model, and add a scope to plot the value of the output b, as in the following figure.



Save the model as Lab3_3. Simulate the system; you should see the variable b incremented by 1 each time the block "Loop" is entered. Change the keyword "entry:" in the Loop block with "during:" and "exit:", and repeat the simulation. What happens?
Hint: the result with the keyword "during:" depends on the simulation step time.

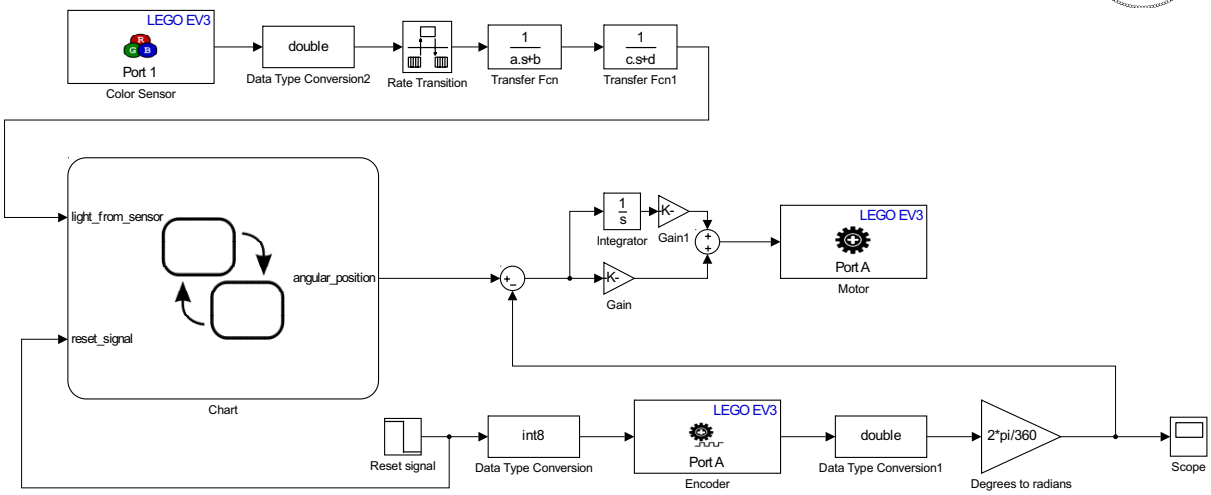
4)
Now let's write an automaton that reads takes a sequence of light readings at evenly spaced angles. Open a new model, insert a Stateflow chart and draw the following diagram.



Add the inputs "reset_signal" and "light_from_sensor" (Chart->Add Inputs & Outputs), and the output "angular_position".
We need to define a global variable, visible from all elements of the model, containing the number of readings that we want to execute. Go up to the level of the main model and select Tools -> Model Explorer -> NameOfTheFile; click on the Callbacks panel, select InitFcn (this is a function that is run at the beginning of the model, we will use it to define a global variable setting the number of readings and accessible throughout the model). Write "number_of_readings = 6.
Then, we must declare all the variables used by the Stateflow chart, and specify their properties:

- Go back to the chart (double click on the stateflow block) and select Chart-> Add Other Elements-> Local Data
- Add the local variable "i" with size : 1
- Repeat as above, and add the local variable "delta_angle" with size : 1
- Repeat as above, and add the local variable "light" with size : number_of_readings. Tick the checkbox "Variable size" for this variable.
- Select Chart -> Add Other Elements -> Constant and add the constant number_of_readings (this way the Stateflow chart has access to the global variable number_of_readings which we defined in the init function)
- In view-> model explorer -> model explorer-> chart, select scope: constant for number_of_readings, and set Initial value : number_of_readings (this gives it the value assigned by the initialization function we defined before).

Now go up to the main model level and connect the chart to the models we prepared in Lab2_6 and Lab3_2, as in the following figure.

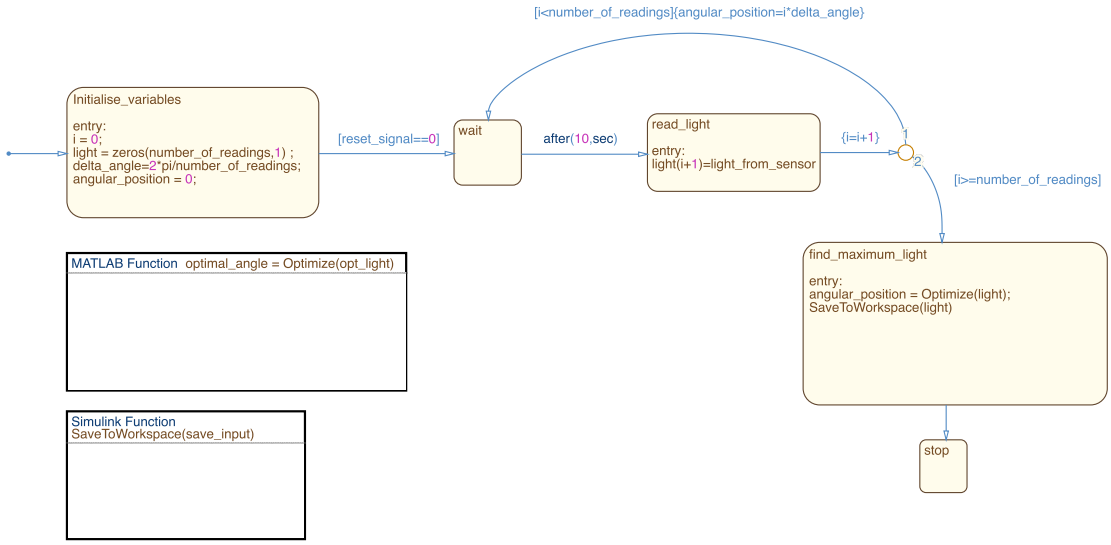


Save the model as Lab3_4. Notice the Rate Transition block added in the color sensor signal line. This serves to translate the discrete time variable returned by the color sensor into a continuous time variable suitable to feed the transfer functions. The block is simply a zero-order hold, that is, a memory which reads its input value at discrete times and keeps the output constant in between readings.

Run the simulation on the Lego brick and check if it places correctly the sensor at equally spaced angles.

Session 4: optimization

1)
Load the last model we built in Session 3, modify its chart as in the following figure, and save it as Lab4_1. The MATLAB Function and Simulink Function blocks can be dragged in the chart from the left-side menu.

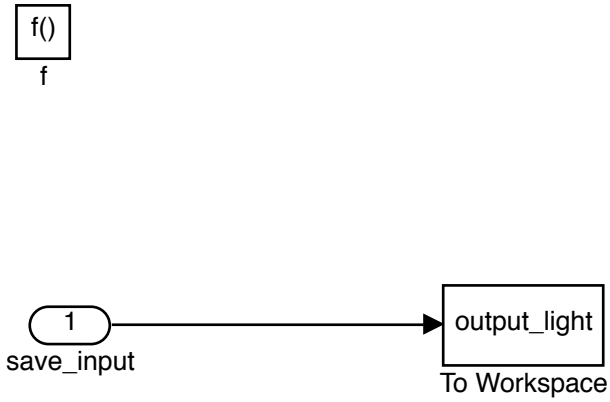


For the moment, double click on the Matlab Function (this will open the function in the standard Matlab editor) and edit it as follows.

```
function optimal_angle = Optimize(o_light)
        optimal_angle = 0;
end
```

This function always returns 0, we will change it later to compute the direction of maximum brightness.

First, we want to have an idea of how the light field looks like around our system. To do so, we need to use the sensor to collect some data, and save it to the workspace. We can do this using the Simulink function. Click on the function name and call it "SaveToWorkspace(save_input)". This defines a function "SaveToWorkspace" which takes the argument "save_input".
Now double click on the function and you will see the argument as a simulink block. Connect it to a To Workspace block as in the following figure.



Double click in the output_light block in the function, and set "format: structure with time" (must be like this for all external mode exports). Double click on the argument (save_input) block and select "signal attributes -> port dim: number_of_readings" and "variable size: yes".
The state "find_maximum_light" is calling this function on entry, this will save the value of the vector "light" in the workspace with name "output_light".
Changing the initialization constant number_of_readings to 24, collect a vector of readings and plot it against the angle:

```
>>angles = 0:2*pi/number_of_readings:2*pi*(number_of_readings-
1)/number_of_readings;
>>plot(angles,output_light);
```

(Remember that each reading requires around 10 seconds, so this will take a while!)
The result should be a roughly unimodal distribution, that is, a function with a single peak (like a Gaussian distribution). We need to find a way to reconstruct such a function, and find its maximum, using as few readings as possible, in order to spare time and energy.

2)
All is left to do is to write the function Optimize to find the angle of maximum brightness. To make the best possible use of the data, we can assume that the light field has a known shape, and describe this shape as a function $f(x,p)$, where p are the parameters of the function. Then we can set up a *parameter identification* problem:

$$\min_p [f(x,p) - \text{light}(x)]^2$$

where light(x) is the value of brightness measured at the angle x. Remember, this is the square of a vector, so it reads as the scalar product of the vector with itself.

Before writing the algorithm in the Stateflow automaton let's tune it through some trial and error in the Matlab command line. Let's try to use a sinusoid of period 2pi as our function $f(x,p)$. We have that

$$a + b\sin(x) + c\cos(x) = a + d\sin(x+\varphi)$$

that is, any sinusoid $a + d\sin(x+\varphi)$ with phase φ and mean value a can be written as the sum of a sine and a cosine with phase 0, plus a constant. Thus, our optimization problem is

$$\min_{\{a,b,c\}} [a + b\sin(x) + c\cos(x) - \text{light}(x)]^2.$$

If we call L the vector light(x), call S and C the vectors $\sin(x)$ and $\cos(x)$ evaluated at the same angles x, and call 1 a vector of ones of the same size as L, C, and S, the optimization problem becomes

$$\min_{\{a,b,c\}} \left[[1\ S\ C] \begin{bmatrix} a \\ b \\ c \end{bmatrix} - L \right]^2 =$$

$$\min_{\{a,b,c\}} \left[[a\ b\ c][1\ S\ C]^T [1\ S\ C] \begin{bmatrix} a \\ b \\ c \end{bmatrix} - 2[a\ b\ c][1\ S\ C]^T L + LL^T \right]$$

We know that the optimality condition for a function $h(p)$ is $\frac{dh(p)}{dp}$=0, and we can apply this condition to the objective function $f(p)$ above, with $p = [a\ b\ c]$. We have

$$\frac{df(p)}{dp} = 2[1\ S\ C]^T [1\ S\ C] \begin{bmatrix} a \\ b \\ c \end{bmatrix} - 2[1\ S\ C]^T L = 0$$

Solving for [a b c] we obtain

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = [[1\ S\ C]^T [1\ S\ C]]^{-1} [1\ S\ C]^T L$$

Use this formula to compute the coefficients a, b, c that best interpolate the light data you collected, and plot the result to test if it is good enough. First, prepare a vector with the values of the angular positions:

```
>>angles=[]
```

Then, prepare the matrix $[1\ S\ C]$ (which we call M):

```
>>M = ones(number_of_readings, 3);
>>M(:,2)=cos(angles);
>>M(:,3)=sin(angles);
```

Then compute [a b c]:

```
>>optimal_coefficients = (M'*M)^(-1)*M'* output_light
>>a = optimal_coefficients(1);
>>b = optimal_coefficients(2);
>>c = optimal_coefficients(3);
```

Now plot the brightness values measured by the system, and hold the figure

```
>>plot(angles, output_light,'ro')
>>hold on
```

And plot the function a +b*cos(x)+c*sin(x) to see if it approximates well the coefficients

```
>>x = linspace(0,2*pi,100);
>>y = a+b*cos(x)+c*sin(x);
>>plot(x,y)
```

Repeat the process above reducing the number of brightness measures to 12, 6, and then 3 (take a subset of measures from the vector output_light, for example take one every two elements, and regenerate the angles vector accordingly), and see what is the minimal number of measures you can use to obtain a good approximation of the real brightness distribution.

Once you have found a good compromise between number of readings and precision, you are ready to write the Stateflow function Optimize.

Double click on the funciton Optimize(o_light) to change its code. Rewrite the function's code so that it computes [a b c] as we did before, and generates the vectors

```
x = linspace(0,2*pi,100);
y = a+b*cos(x)+c*sin(x);
```

then type

```
[~,i]=max(y);
opt=x(i);
optimal_angle=opt(1);
```

to assign to the output "optimal_angle" the value of the angle x where y reaches its maximum. Save the model as Lab4_2.
Run the model on the Lego brick to test if it works.
Good luck!