Chapter 16
Systolic Arrays

16.1 A brief Introduction

The CPU architecture you have encountered so far consists of a single processing element 'PE' (think 'core') with nearby code and data memory. All instructions from a running program are fetched from code memory and execute on the single PE. Sure, some CPU chips are multi-core, but this architecture remains invisible to the programmer who just sees his procedural code running on a single CPU.

Systolic arrays offer a completely different architecture and programming paradigm. Perhaps the simplest is shown in Fig.1. There is a chain of processing elements (PEs) each of which is able to run a different program. Data enters at the top, passes down the chain and exits at the bottom. Think production line where each stage adds its own parts. Each PE will start its processing when data arrives at its input, so we have an asynchronous system.

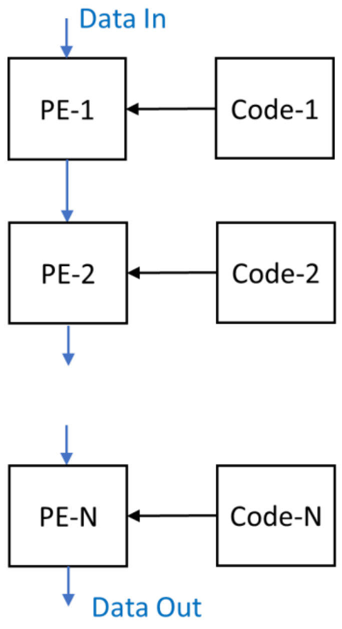


Figure 1A simple linear systolic array

The name 'systolic' is inspired by the human heart 'systole' where the heart contracts and pumps blood through our pipeline.

This chapter explores a particular systolic array implemented on a chain or ring of Arduinos to solve the problem of how to implement a segmented robot. You have already seen our model of a Hexapod (Chapter 9), here we extend some ideas presented there to a robot comprising N segments.

16.2 Topologies, Applications and Properties

What are the possible topologies of a systolic array? Well, the PEs can be viewed as nodes in a network, so systolic array topologies will closely resemble network topologies. Some examples are shown in Fig.2. There is (i) the linear topology

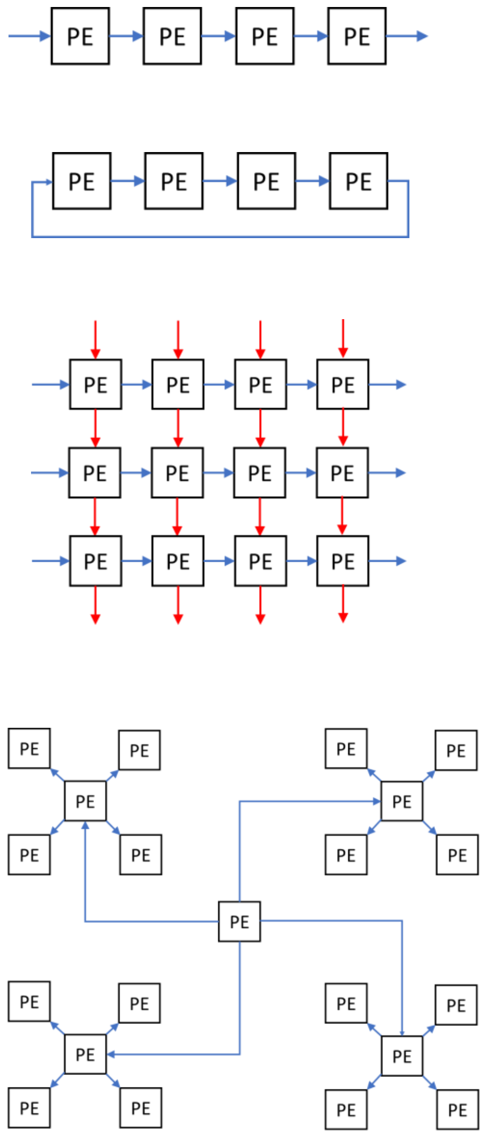


Figure 2 Some Systolic Array topologies. From the top (i) line (ii) ring (iii) 2D array, (iv) Binary tree.

which can be enhanced by a tail-to-head connexion to form (ii) a ring. Then there is a 2D interconnexion matrix (iii) and even a binary tree (iv). I guess you can work out what other topologies are possible.

Some important applications run efficiently on systolic arrays. Many image processing applications involve convolution with a smoothing or edge-detection kernel. These are repetitive and can be easily spread over a load of PEs which run identical code. Digital signal processing also involves similar operations, these are very important applications of computing. Matrix operations (such as 3D transformations in video games) can also be mapped onto systolic arrays. But there are some issues, notably the integration with conventional processing architectures, where data bandwidth may be a problem; systolic arrays expect data to arrive and leave with extremely high rates. Also, there is the issue of developing algorithms which is different from the conventional procedural paradigm. This is usually done manually and requires considerable effort.

However, we have identified one class of interesting computations which can be easily mapped onto a systolic array, indeed the systolic array is the ‘obvious’ solution to this class. That’s the case of segmented robots.

16.3 Systolic Array Communication

There are various ways of getting Arduino’s to talk to each other, Bluetooth could be a good choice, however BLE-enabled Arduinos can be relatively expensive, so are not best suited for an N-segment robot. We must turn to other solutions, and even become creative.

16.3.1 I^{2}C bus communication

This is a 2-wire serial bus originally designed for communication between integrated circuits. One wire SCL carries a clock signal, and the second SDA carries the data. Each device on the bus has a unique address, and at any one time only one device transmits data to another, this is often referred to as a ‘master-slave’ architecture. In our case of the segmented robot, each segment has its own Arduino PE and a unique address.

The code for each segment is organized around a Finite State Machine (FSM) which coordinates all aspects of both communication and computation. The code for each PE has the structure shown in Fig.3. The three states are Compute, Transmit and Receive. In Compute we solve some maths equations (discussed later), in Transmit we send the result across the I2C bus to the next PE, which is in the state Receive, ready to grab the data. In this state, the PE actually gets the data using the Receive Event call-back function which is part of the I2C-Wire protocol.

When the chain is initialized, all segments start in state Receive, except the Head which is in Compute, to get the processing chain started. This is indicated by the green line in Fig.3. When the computation is complete, then we enter state Transmit and send the result to the Next PE, and then we transit to state Receive where we sit quietly until another PE sends a message to us and wakes us up. Then we return to state Compute.

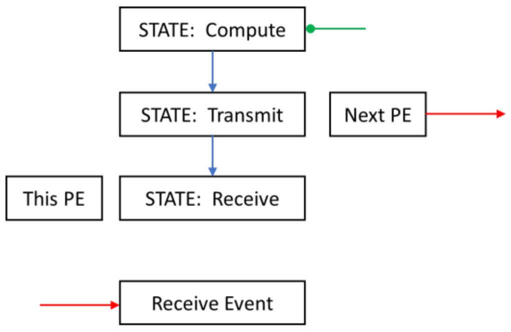
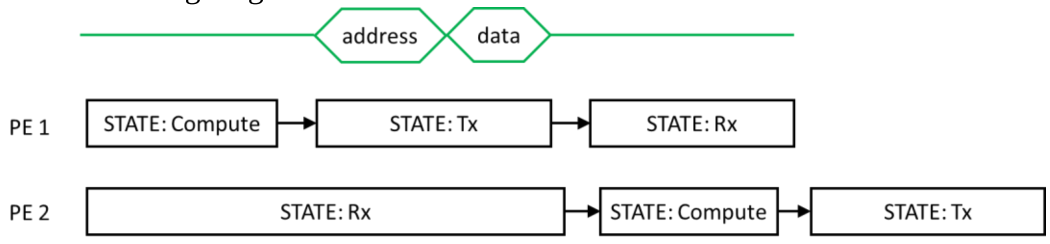


Figure 3 The PE state machine.

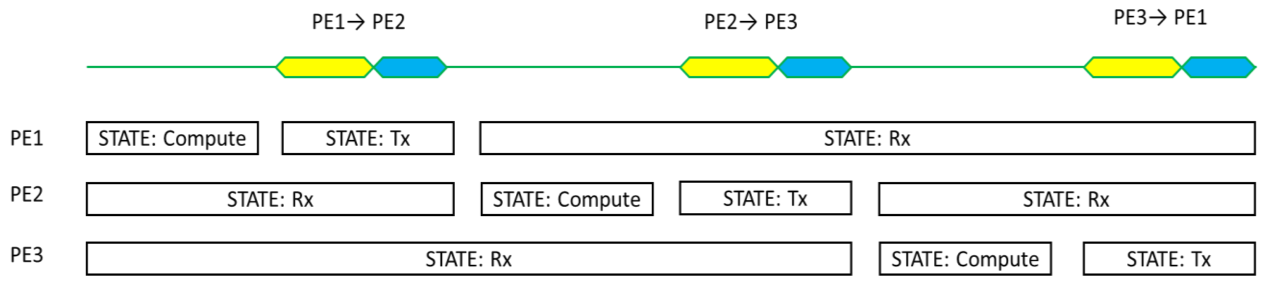
The TAIL segment is a little unique since it transmits to the HEAD when ready. The purpose of this is simply to get the Head to transit between Receive to Compute, to get the next wave of processing started.

We can more usefully show the PE states and intercommunication as a ‘timing diagram’ as we saw for the Vivado testbench waveforms when we were synthesizing digital circuits. The diagram below shows the building blocks of such a timing diagram.



PE1 starts in state Compute, and when this is compete, it transits to state Tx (transmit). The top line shows the I2C bus, and you can see that PE1 first sends the address of PE2 (where it wants to send its payload) then it sends the payload data. Meanwhile PE2 is in state Rx (receive) so it is able to gobble up the incoming data. It then transits into its Compute state, where it processes this data, and in turn transits to state Tx where it sends the data to the next PE.

Now let's consider the specific architecture of three PEs arranged in a loop (so the third feeds back into the first). In this case, the timing diagram will look like this, where I2C addresses are shown as yellow, and data as blue. PE1 starts in state Compute and PE2 and 3 in state Rx (you probably got the idea that state Rx is a waiting state where the PE just hangs around). When PE1 has finished computing, it asserts PE2 as its address and puts its data onto the I2C bus. PE1 then reverts to state Rx. Meanwhile PE2 transits to state Compute and it does its processing. When complete, it asserts PE3 as its address and puts its data onto the I2C bus which is received



by PE2 which has so far been in the waiting state Rx. Then PE2 reverts to state Rx and waits for further incoming. At the same time PE3 starts its processing and when complete, asserts PE1 as its address and puts its data onto the I2C bus. It will go into state Rx and then the cycle will start again with PE1 transiting into state Compute.

Looking at the above timing diagram, you might conclude that this computing approach is incredibly inefficient, and you would be right since it appears that the PEs spend most of their time in state Rx, waiting, listening for incoming data. This is a direct consequence of using the I2C bus where only one PE may put a data packet on the bus at any time.

16.3.2 Analogue Voltage Communication

Arduinos have analogue-to-digital (A-to-D) convertors which can measure an input voltage and convert this to a digital value, so we use this as the input to a PE. Unfortunately, they don't all have digital-to-analogue (D-to-A) convertors on board, but we can add such a chip. The D-to-A convertor converts the digital result of the PE computation to an analogue voltage, which is the output of the PE. The idea shown in Fig.4. Each PE receives input data as an analogue signal on a single wire (red). The Arduino A-to-D convertor digitizes this and feeds it into the Compute part of the PE. The output of Compute is then converted to an analogue signal via the D-to-A convertor, which is output to the following PE.

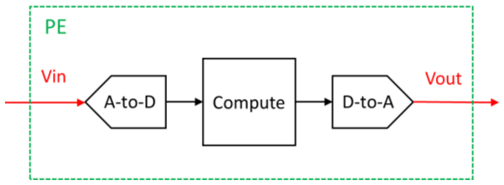


Figure 4 Analogue PE communication. Red shows analogue data on a single PE-PE wire.

So our exemplar chain of three PEs when connected in this manner looks like the circuit shown in Fig.5. Note there is no need to feedback from the last PE to the first since the analogue signal changes in real time, and there is no need to send back a synchronization signal as in the case of I2C communication. This is an important point, using analogue voltage communication we can obtain (almost) real-time processing since there are no communication delays. This model is closer to actual biological segmented systems such as snakes and fish, both have inspired segmented robots,

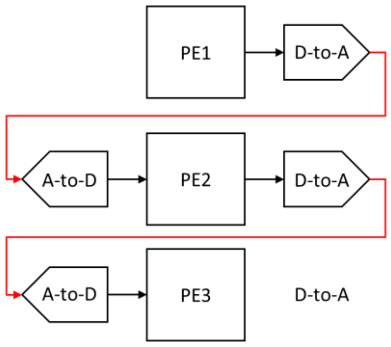


Figure 5 Chain of three PEs coupled using analogue voltages.

16.3.3 Pulse Length Communication

It is straightforward to program the Arduino to produce pulses of definite time duration on an output pin, and also to measure the duration of pulses arriving at an Arduino input pin. We can use this to communicate data if we convert the value of a data variable to the length of a pulse, so that the pulse-length is proportional to the value. So a PE will emit a pulse of this length, and the receiving PE measures the length of the incoming pulse and can recover the value of the data

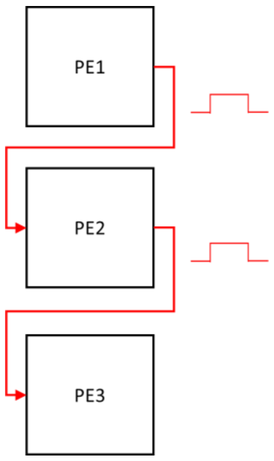


Figure 6 Segments coupled using digital pulse length.

variable. Our chain of three segments now appears as shown in Fig.6 where, again, we need just a single wire for communication.

Producing pulses is straightforward, we just use some code like this on the sending PE.

```
duration = 100 + 100*var;

digitalWrite(8, LOW);
delayMicroseconds(100);
digitalWrite(8, HIGH);
delayMicroseconds(duration);
digitalWrite(8, LOW);
delayMicroseconds(100);
```

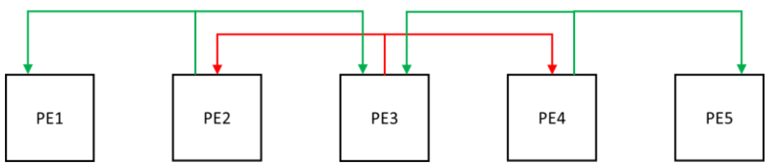
Measuring pulse length on the receiving PE needs a little more thought. A simple solution is to use the Arduino pulseIn function which returns the duration of an input pulse in microseconds.

```
duration = pulseIn(7, HIGH);
inputVal = ((float)duration - 100.0)/100.0;
```

Note that the functions which translate between data value and pulse lengths are inverse. While the use of pulseIn(…) does work it does have a flaw, it is a *blocking* function which means that it does not return until it is complete. This gives an issue when a PE needs to receive two pulse inputs (e.g., from its neighbours) simultaneously. In this case we must use interrupts.

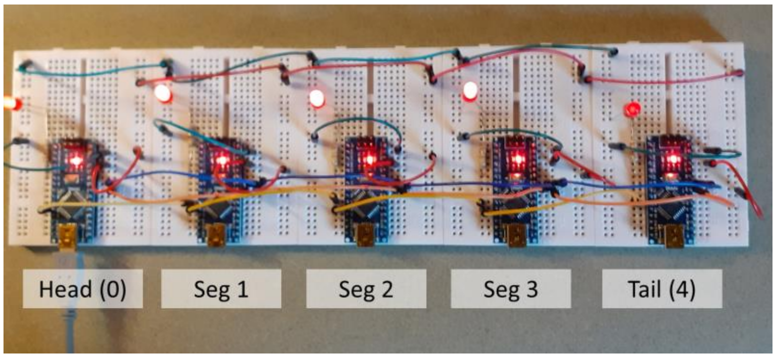
16.4.Bi-directional communication

Some applications of systolic arrays require segments to communicate with both their anterior (upwind) segment and their posterior (downwind) segment. This is possible using both the analogue and pulse length communication protocol as shown below. I2C bidirectional communication is difficult since only one segment can write to the bus at any one time. The diagram below shows you the idea. As mentioned above, if we choose pulse-length communication, we cannot use pulseIn(…). We shall discuss this later in the Lamprey section (16.7).



16.5 An Arduino Multi-Segmented Robot

Here we shall discuss a multi-segmented robot consisting of a Head and N-segments that follow. We shall use the neural circuit model introduced in Chapter 9.
The figure below shows a chain of 5 PEs, each is an Arduino Nano device with the Head and Tail segments labelled. Processing starts at the head and when complete, data is sent down the array where it is processed by each PE. This chain



is actually a *ring* since the Tail communicates with the Head.

Connexions for both I2C and pulse length communication protocols are implemented. Pulse length uses digital pins 8 (output) and 7 (input). The I2C uses analogue pins A4 (SDA) and A5 (SCL). The SCL signal is a serial clock and SDA is serial data. The values of one data variable on each PE is indicated by the brightness of a LED which is controlled by pulse-width modulation.

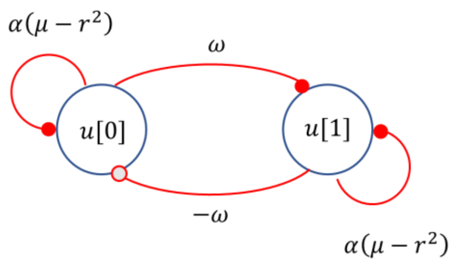


Figure 7 Neural circuit for the robot head. Full circles are excitatory connections, empty circle is inhibitory. Note the symmetry of this circuit.

The neural circuit we use has two different types of PE. The Head comprises an oscillator (as in Chapter 9) and each body segment consists of a phase-delay neural circuit. The theory behind this is discussed in Chapter 9, here we shall focus on the code implementation. The head oscillator is formed by two neurons, it's a Hopf oscillator shown in Fig.7

The dynamics of this oscillator is expressed as a pair of ordinary differential equations coded like this which corresponds to the circuit shown in Fig.7.

```
r = sqrt(u[0] * u[0] + u[1] * u[1]);
dudt[0] = alpha * (mu - r * r) * u[0] - omega * u[1];
dudt[1] = omega * u[0] + alpha * (mu - r * r) * u[1];
```

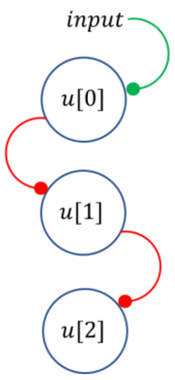


Figure 8 Three neurons forming a phase-delay PE.

The output from the Head PE is just u[0]. The remaining body segment PEs are phase delays, each PE containing 3 neurons.

To obtain the phase delays in the PEs of the remaining segments we need a chain of three neurons, since a single neuron can produce just under 90 degrees of shift, and we may need much more. The code to obtain this is shown below where the constants k and tau are determined by the actual phase shift, we require.

```
dudt[0] = (-u[0] + k * input) / tau;
dudt[1] = (-u[1] + k * u[0]) / tau;
dudt[2] = (-u[2] + k * u[1]) / tau;
```

The input to the neurons is shown, the output comes from neuron u[2]. If you look at the code in detail, you will see that, with the exception of the Head, the output u[2] passes through a **tanh(…)** function, Fig.9. This friendly function ensures that the output is always in the range -1 to +1 for any input. Knowing the range is useful when we may pass the signal on for further processing, e.g., to drive a motor.

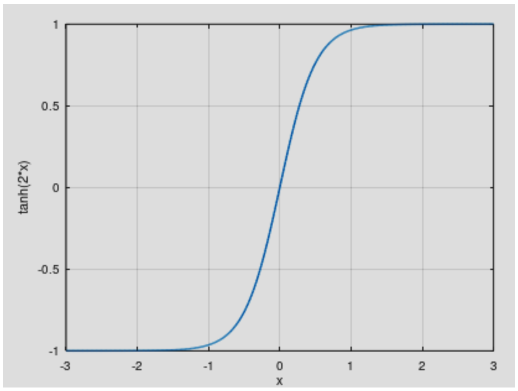
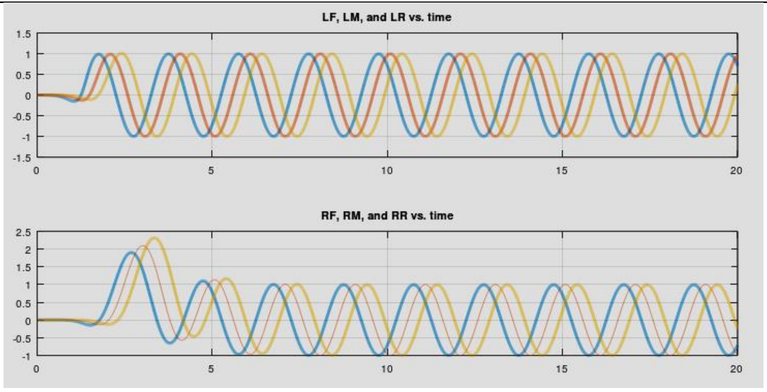


Figure 9 Tanh function.

To get an idea of the behaviour of our Arduino systolic array, we can 'cheat' and solve the problem using Octave. The plots below show the behaviour for three segments (one head and two body segments) where the phase shift has been set to 60-degrees.



There's lots of information in these plots. First, we can see periodic behaviour, thanks to the Head oscillator (blue line). Then the periodic behaviour passes to the first body PE (red line) and then to the second (orange line). You can see a clear phase difference between the PEs, and if you look carefully at the plot, you will convince yourself this is 60 degrees.

Note also that all signals are in the desired range of -1.0 to +1.0 as discussed above. Finally you will note that it takes around 5 second for the circuit to settle into a periodic behaviour; this is called a 'transient'.

On the actual Arduino PE's this behaviour can be seen through the brightness of the LEDs which is proportional to the PE signal. This is done through standard pulse-width modulation.

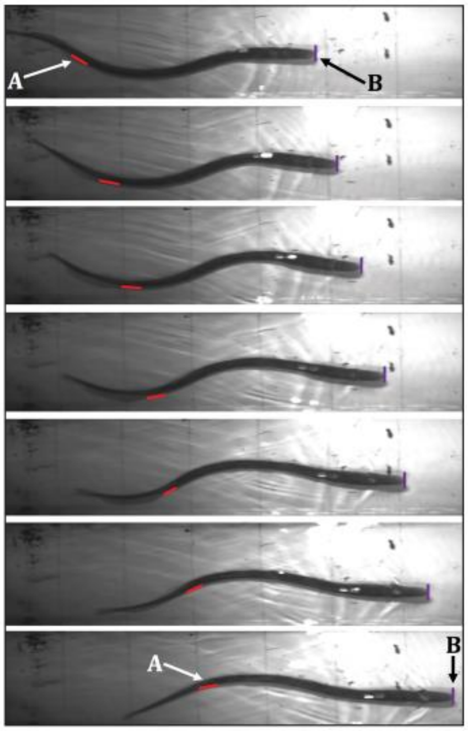
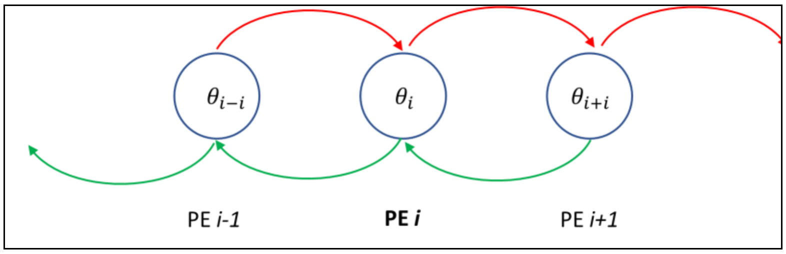


Figure 9 Screen capture images of a swimming sea lamprey [Photo attribute request]

16.6 The Lamprey Model.

The Lamprey is a primitive vertebrate which, without any fins, swims with a snake-like motion by deforming its body in a very special way. There is a propagating wave of movement down the backbone segments. This critter has been the subject of a lot of mathematical modelling and analysis, computer simulation and has been used as inspiration for the development of swimming robots.

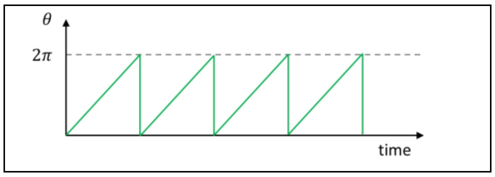
Here we discuss the 'standard' neural circuit model for the lamprey shown below where we have drawn just three neurons (PEs) out of a chain on N PEs.



Consider the middle (i'th) PE. We can see that it is connected to its *anterior* PE (*i-1*) and also to its *posterior* PE (*i+1*), this means the Head is at the left and the Tail is at the right. The state variable of each PE is the angle θ which is called its 'phase'. The phase of each segment increases linearly with time, and is described by the ordinary differential equation (ODE)

$$\frac{d\theta_i}{dt} = \omega_i \qquad (1)$$

where ω_i is the frequency of oscillation of segment PI-*i*. What does this mean? Well interpreting θ as an angle, when this increases linearly and reaches 2π rads (or 360 degs), then it 'wraps around' and becomes zero again, and then starts to rise, a bit like this.



Now we need to turn to the coupling between the PEs, as mentioned above we need to couple to both the anterior and posterior PEs For sound theoretical reasons we choose to model the coupling as a function of the phase difference between neighbouring segments, i.e., $(\theta_{i-1} - \theta_i)$ for PE-i and its anterior. We choose this function to be periodic and in particular a sine function. The coupling terms therefore look like this.

$$a_U \sin(\theta_{i+1} - \theta_i) \qquad (2a)$$

$$a_D \sin(\theta_{i-1} - \theta_i) \qquad (2b)$$

Here (2a) refers to the coupling with the posterior PE (the literature refers to this as 'upward'), here a_U is the coupling strength, and (2a) refers to coupling with the anterior PE.

Now we combine eqs (1) and (2) and build up the ODEs for our PE chain.

$$\frac{d\theta_i}{dt} = \omega_i + a_D \sin(\theta_{i-1} - \theta_i) + a_U \sin(\theta_{i+1} - \theta_i) \quad (3)$$

with special cases for the Head,

$$\frac{d\theta_1}{dt} = \omega_1 + a_U \sin(\theta_2 - \theta_1)$$

and for the Tail

$$\frac{d\theta_N}{dt} = \omega_N + a_D \sin(\theta_{N-1} - \theta_N)$$

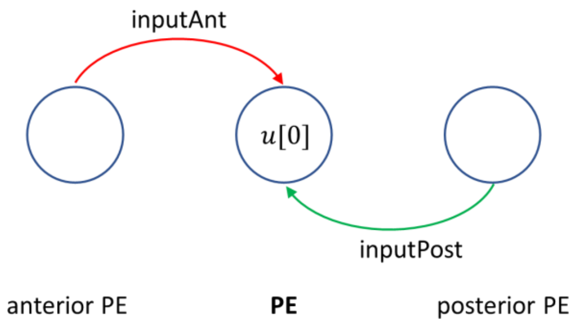
It's quite straightforward to convert these equations into code which runs on the Arduino, for the Head, Body and Tail we have, respectively

```
dudt[0] = omegaHead + au*sin(inputPost - u[0]);
```

```
    dudt[0] = omegaSegm + ad*sin( inputAnt - u[0] )
                                + au*sin( inputPost - u[0] );
```

```
dudt[0] = omegaTail + ad*sin (inputAnt - u[0]);
```

You will at once notice a fundamental difference between the model expressed in the ODEs (3) and the above code. The ODEs refer to subscripted state variables θ_i, while each PE has one ODE u[0] and inputs. Let's redraw the above structure diagram from the point of view of coding.



Here the circles (neurons) are incarnated as an individual Arduino on a PE, and the connexion arrows are live electronic signals.

So you see the lamprey model requires bi-directional communication which can be coded using either pulse-length or analogue voltage approaches.