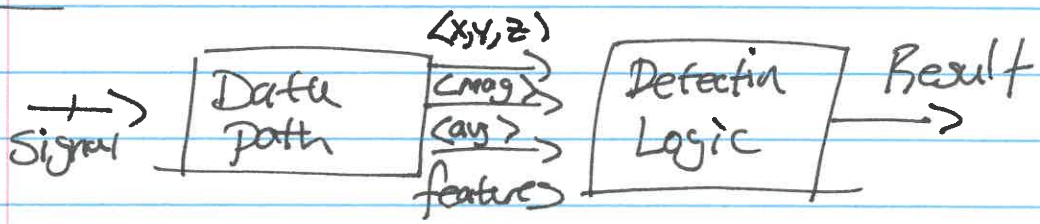
From Algorithms to Code

10/4/21



* data path expressed as series of equations
  Logic as rules ~~for~~ as logic, FSM, flow chart....

$i_0 \equiv$ previous value (i_{n-1})
$i_1 \equiv$ current value (i_n)
$i_2 \equiv$ future value (i_{n+1})

needs no memory → $mag_i = \sqrt{x_i^2 + y_i^2 + z_i^2}$

needs previous value → $diff_{i,j} = mag_{i-1} - mag_i$

no memory → $Square_k^{i,j} = diff_{i,j} \cdot diff_{i,j}$

Needs N results → $arg_l = \sum_{k}^{k+N} Square_k$

* are i, j, k, and l the same or different?
* how much storage is needed for each steye? How does the computer hold this information? Is it bounded?

Process

- for each feature, establish how much memory or historical information is needed. The needs of later pipeline steyes often dictate how much is stored by intermediate variables

- may assign each steye more memory than required for each of calculation.

- Now that you've allocated space for each steye, how is data read in?

```
int values[10];
for (int i=0; i<10; i++)
{
    values[i] = newData();
}
```

what's wrong w/ this code?

* What happens when the data source contains more than 10 results? What if it contains 10^6 or is infinite?

=> key question is how much data is needed for each steye? Do need all 10^6 values or just the last 10?

```
const int BUFFER=10;
int values[BUFFER];
for(int i=0; i<10^6; i++)
{
    values[i % BUFFER] = newData();
}
```

i always holds position of "newest" data point. i-1 is the next oldest.

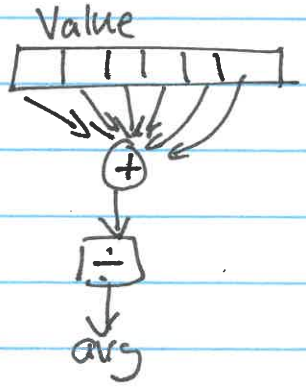
use the modulus operation to ensure writing to the array never overflows the bounds.

→

these would be on different lines

* Now lets combine all these things; what if a function wanted a 10 way average across a chunk data set:

```
int value[BUFFER];  const int BUFFER=10;  int idx=0;
While(data_available)
{
    value[idx % BUFFER] = newData();
    float average=0;
    for(int j=0; j< BUFFER; j++)
    {
        average += value[j];
    }
    average /= BUFFER;
    idx++;
    cout << "Latest average is..." << average << endl;
}
```



Since using while loop need own index variable

* features of this code: Works regardless of the buffer length. Would always be slow as iterates over value[] time but this can be reduced single operation if replaced values are tracked.

- Now, consider what happens when operations need to be chained between different arrays. What if the moving average was on the difference of two points and not the raw value itself?

assume some constant length

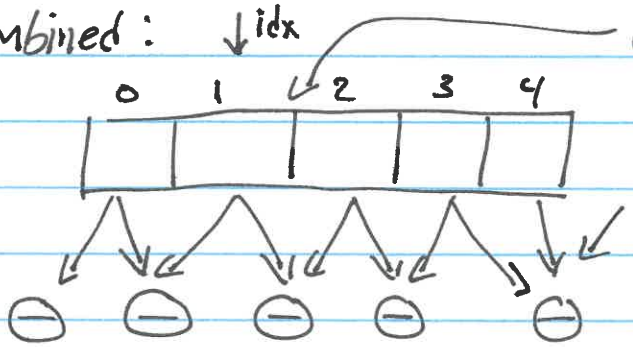
```
int values[LEN]; int idx=0;
while(data_available){
    values[idx%LEN] = newData(); float sum=0;
* must iterate over LEN-1 elems   for(int j=0; j<LEN-1; j++){
        sum += [idx%LEN] - values[(idx-1)%LEN];
```

grab current value pointed by idx

grab "previous" value by idx-1

* finish out loop as before

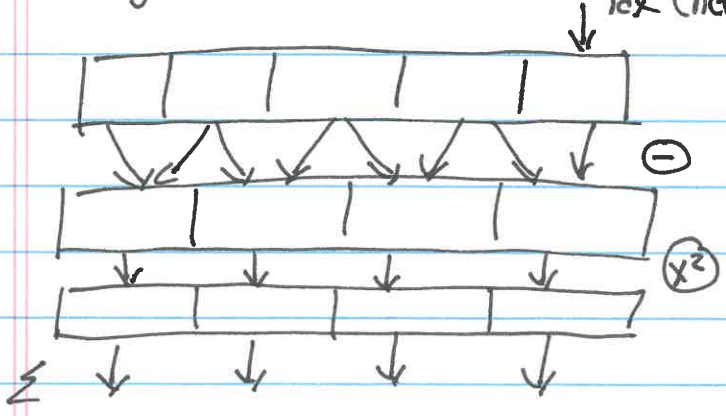
Process can be drawn to visualize how elements are combined:



don't take diff here as values are not adjacent. (idx+1) may be far in the past

values[5]

So now, return to the faststep pipeline and see how all the elements map out: Assume a simple moving average over 5 elements



* only have four values at then end because one value lost due to diff. Need at least N+1 at start

Assuming a moving average of 5 points, need at least 6 values in "input" buffer and 5 elsewhere.

```
const int LEN=5;
int values[LEN+1]; int diffs[LEN]; int squares[LEN];
int idx=0;
while(data_available){
    values[idx%(LEN+1)] = readData();
    diffs[idx%LEN] = values[idx%(LEN+1)] - values[(idx-1)%(LEN+1)]
    squares[idx%LEN] = diffs[idx%LEN] * diffs[idx%LEN];
    for(int j=0; j<LEN; j++){
        sum += squares[j]
    }
    float avg = sum/LEN(float)
```

float sum=0

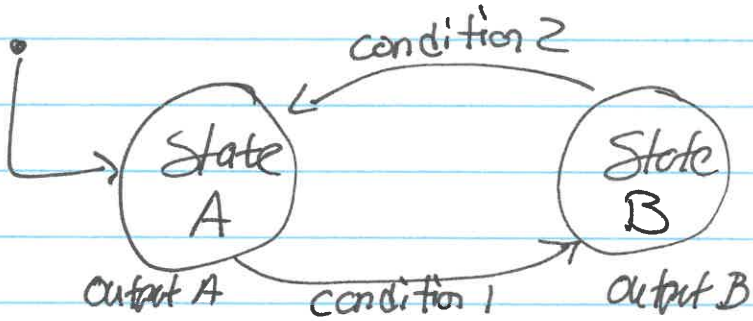
will need to cast as float.

- The sequences above only show how to produce a single result. They may need to be extended if the decision making or logic process requires multiple outputs to render a result.

The previous examples showed a method for transforming a data pipeline into a series of code snippets. After following this stage there is another one focused on decision making based upon the data/results generated.

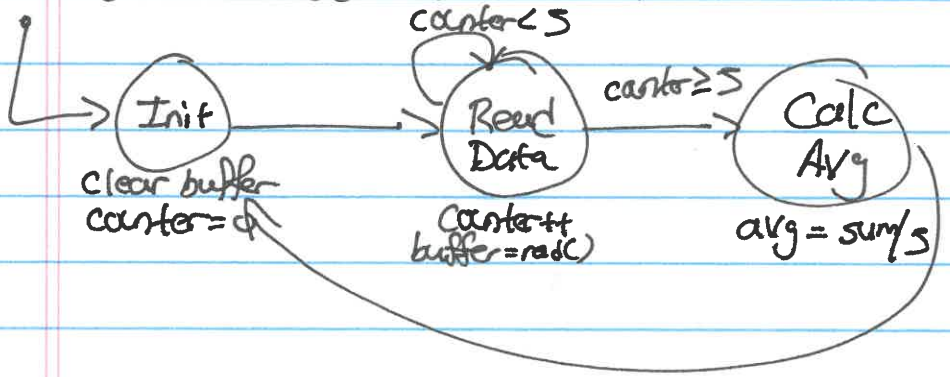
An approaches logic or decision making can be represented in a variety of methods. Two most common are finite state machines (FSM) and flow charts. We will cover FSMs as they are more general and widely used within embedded systems.

An FSM consists of three major parts: a state, a transition, and an action. The system is assumed to always be in a certain state that it moves between based upon input information and the defined state transitions. In each state the system produces one or more outputs.



FSMs can be represented at many levels of abstraction; select the one most useful for your application.

Imagine a machine that would read in five data points and then determine if there average was above a certain threshold.



The machine above has three states. The first prepares a buffer to hold data and initializes a counter. The second reads in new data until there are 5 elements in the buffer. Once the number of elements are readed the system calculates the average before repeating the process again. This logic could be implemented as the small snippet below.

```
While(true) {
~~int counter=0;~~ int buffer[5]; // create variables
memset(buffer, 0); // clear buffer
for(int i=0; i<5; i++){
    buffer[i] = readData(); }
~~float avg=~~ float avg = sum(buffer) / 5;
```

This program loosely implements the FSM but this process can be more formal (and is advised) using state variables w/ switch/case statements.

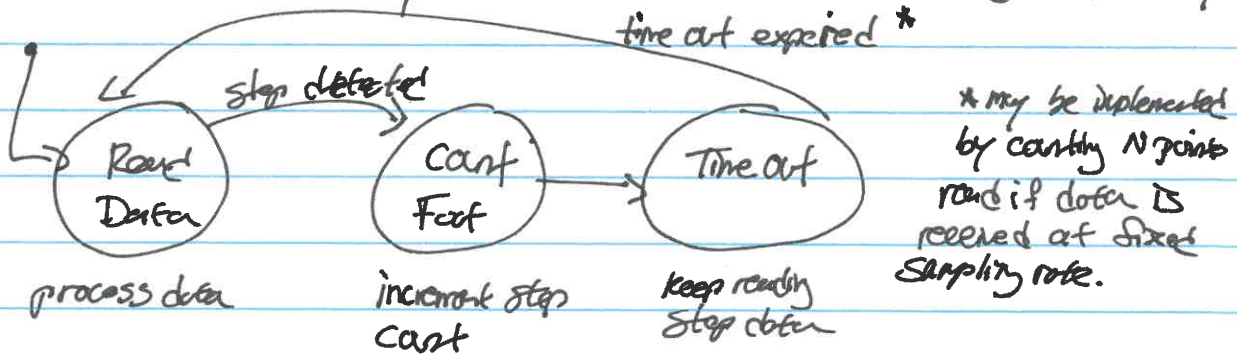
```
State currentState = Init;  State nextState = Init;
Int buffer[5]; int counter = 0;
While (true){
  switch (currentState):
    case Init :                    // clear buffer and setup mem
        memset(buffer, 0);
        counter = 0;
        nextState = ReadData;  // next state is always ReadData
        breale;
    case ReadData :                // read 5 elements
        buffer[counter] = readData(); counter++;
        if (counter < 5)           // decide on the next state
            nextState = ReadData;
        else
            nextState = CalcAvg;
        break;
    case CalcAvg :                 // Calculate average
        for(int i=0; i<5; i++){
            sum += buffer[i];}
        avg = sum/5;
        ~~break;~~ nextState = init; break;
    default :
        // warning, unknown state

  current state = nextState;       // update state vars
```

- Some important notes about this FSM, each case must end with a break statement or the code will keep running into the next state.

- each state must take its own actions and update the state variables otherwise the machine will not advance. A default state is always included to catch errors.

- However, this system allows proofs that the machine will advance. And the FSm form can directly map to code.

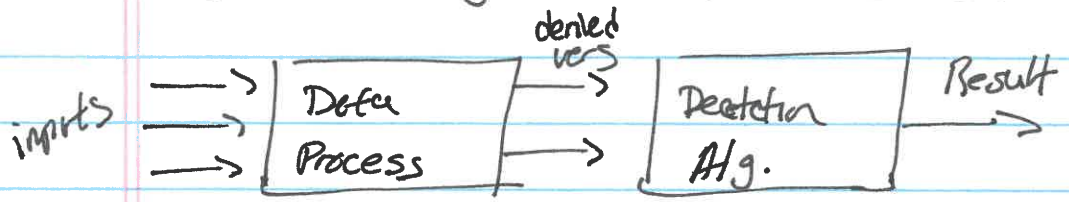
- An FSM is a good approach to model ~~catch~~ time out information in an Algorithm. Consider threshold counting in foot step information. Once a peak/foot step is counted it is unlikely that one will occure again quickly.



This FSM allows the system to keep processing, but not counting, new step information as it arrives. This FSM is of a higher level then previous ones and will require a more complex code (Soften) to model the timing but still can follow the case/switch framework.

* None of these approaches are hard rules that must always be followed. What is important is finding a method that enables one to effectively move from ideas to reliable, readable, and valid code implementations

Both the data processing pipeline and the decision algorithm must work together to render some result.



Within a single program this can become large.

```
int main(){
int array[]; int array2[];...
State currentState = Init; State nextState = Init;
While (true){
    array[i] = readData();
    array2[i-] = .... //process other values
    ;  // do more processing
    switch (currentState):
        case Init:
            ≃
        case StateA:
            ≃
    nextState = currentState;
```

process

decide