Research methodology - general principles of building a software system for data processing.

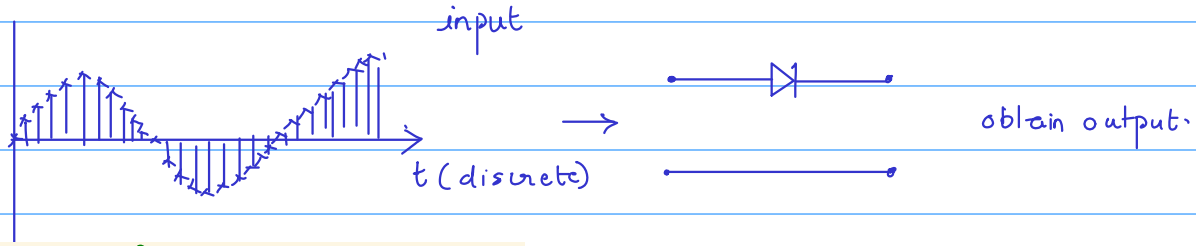
Course outline: Data processing is an essential component of any research venture today. Several data processing tools, software, libraries, and languages exist. In this course, we will explore some of the general principles of setting up a data processing system in a general purpose language (Python). Using these general principles in your own research would lead to well designed systems, with readable code, easier debugging, and reproducible research. This exploration is done via the detailed design of an example.

Lecture 1 Outline:

- Introduction to the course, Procedural programming
- Python (and Matlab) introduction.
- Variables and other data storage mechanisms
- Operations on variables / data processing
- Conditional flow
- Functions & libraries.

Assignment 1:

Simulating a rectifier operating on a sinewave - procedural programming.




```
1 import numpy as np
2 import scipy
3 import pylab as pl
4
5 # Sample times
6 t = np.arange(0, 2, 0.001)
7
8 # Input sine wave
9 f = 50
10 input = np.zeros(len(t))
11 for i in range(len(t)):
12     input[i] = np.sin(2 * np.pi * f * t[i])
13
14 # Simulate rectifier
15 output = np.zeros(len(t))
16 for i in range(len(t)):
17     if input[i] < 0:
18         output[i] = 0
19     else:
20         output[i] = input[i]
21
22 # Visualize output
23 pl.plot(t, input)
24 pl.plot(t + 0.001, output)
25 pl.show()
```

} libraries of functions which we can make use of. When we start out in making data processing systems, try to make use of standard libraries as much as possible without "reinventing the wheel"

→ doing the same operation multiple times.

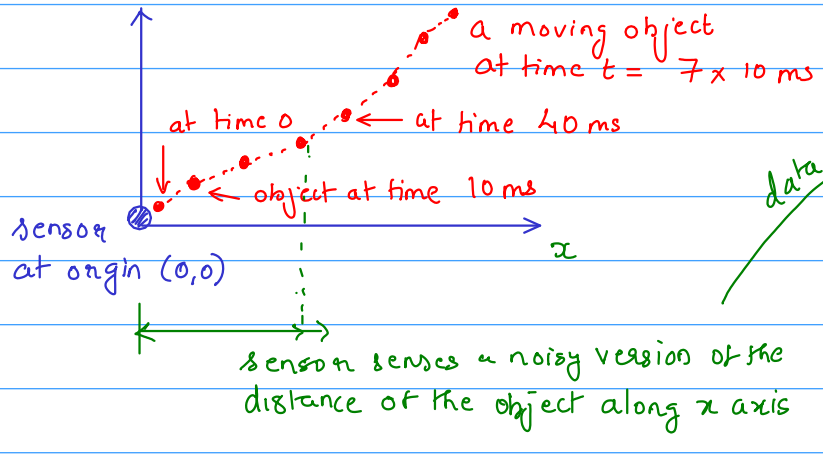
→ use of an inbuilt library function

} use of conditional flow.

} use of inbuilt functions for visualization.

Lecture 2 - an exploration of the general principles via an example.

We will use the following data processing example to illustrate the design of a software system that will help us to explore some good ideas to keep in mind.



data

#	Time	$dx(n)$
0	0	0.5
1	10	1.1
2	20	1.7
3	30	2.5
4	40	3.2
	⋮	⋮

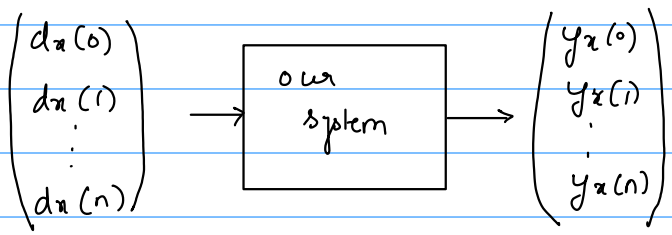
We need to build a system to clean this data. We will use the following algorithm to obtain the cleaned data which we will denote as yx.

$$yx(0) = dx(0)$$

$$yx(n) = \left(yx(n-1) + v \cdot \Delta t\right)(1-\alpha) + \alpha \cdot dx(n)$$

(The motivation for this has been discussed in class).

Our system is a block that can be represented as



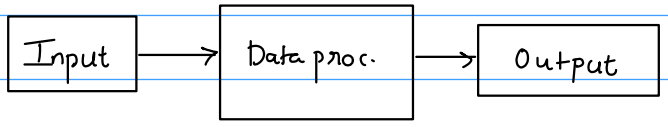
We can make a simple procedure to do this:

```python
import numpy as np

v = 1
delta_t = 0.001

def clean_data(dx, alpha):
    yx = np.zeros(len(dx))
    yx[0] = dx[0]
    for i in range(1, len(dx)):
        yx[i] = (yx[i - 1] + v * delta_t) * (1 - alpha) + alpha * dx[i]

```

but is this the best design that we can have for our system?

In order to design a "good" system it is better to break up your system to subsystems, design those subsystems very well. In our case, it is actually better to use object oriented programming to design systems using this principle.

What are the subsystems you should design?



} design as subsystems so that you have lot of flexibility in doing experiments.

Each block can be represented by a class in Python.

```python
import numpy as np

class Filter:
    def __init__(self, alpha):
        self.alpha = alpha
        self.y = 0;

    def initialize_y(self, d):
        self.y = d

    def process_one_data_point(self, d):
        self.y = (self.y + 0.001) * (1 - self.alpha) + self.alpha * d
        print self.y

    def process_multiple_data_points(self, d):
        for i in range(len(d)):
            self.process_one_data_point(d[i])

    def file_output(self):
        do something for writing y to a file

f = Filter(0.1)
d = [0.1, 0.2, 0.3]
f.initialize_y(d[0])
f.process multiple data points(d[1:3])
```

← Class for the data processing block.

} classes will have a constructor for initialization

Use descriptive variable names, words separated by underscores or CapitalLetters.

→ blocks will have parameters. These are useful for experimentation.

→ write functions that have a well defined objective.

} always test your subsystems

The output block represented as a class

```python
import numpy as np

class Output:
    def __init__(self, type_of_output, filename):
        self.type_of_output = type_of_output
        self.filename = filename

    def produce_output(self, t, y):
        if self.type_of_output == "csv":
            np.savetxt(self.filename, [t, y], delimiter=",")
        elif self.type_of_output == "excel":
            pass
        elif self.type_of_output == "fig":
            pass

o = Output("csv", "temp.csv")
o.produce output([0,1],[1,2])
```

} have multiple options for the output of your data processing system

} again test your subsystems as much as possible.

Lecture 3 – continuation of lecture 2.

Let us now look at the input block. We can assume that input data might come in many formats. Also for testing our code and doing simulation experiments we would also need a simulator for simulating the motion of the object and producing the data as shown in the table. So how should the input block be implemented as a class?

- Should the input block implement a file reader as well as a simulator for data?

maximum flexibility + intermediate data. Simulator → file → input block.
control flexibility by your workflow.

```python
import numpy as np

class Input:
    def __init__(self, type_of_input, filename):
        self.type_of_input = type_of_input
        self.filename = filename

    def get_input(self):
        if self.type_of_input == "csv":
            pass
        self.type_of_input == "excel":
            pass
        # return - what should be returned?
    def get_one_input(self):
        if self.type_of_input == "csv":
            pass
        self.type_of_input == "excel":
            pass
        # return - what should be returned?

```

different file formats for inputs

different ways of obtaining inputs.

The simulator for testing our code or for experimentation.

```python
import numpy as np

class Simulator:
    def __init__(self, v, delta_t, x0, sigmasq):
        self.v = v
        self.delta_t = delta_t
        self.sigmasq = sigmasq
        self.x = x0
        self.times = []
        self.dx = []
        self.actual_x = []

    def simulate(self, number_of_times):
        time = 0
        for i in range(number_of_times):
            self.times = self.times.append(time)
            self.dx = self.dx.append(self.x + np.random.randn() * sigmasq)
            self.actual_x = self.append(self.x)

            self.x = self.x + self.v * self.delta_t

    def file_output(self, output_filename):
        pass

    def run_simulator(self, output_filename):
        self.simulate()
        self.file_output(output_filename)

# How to test this module?
```

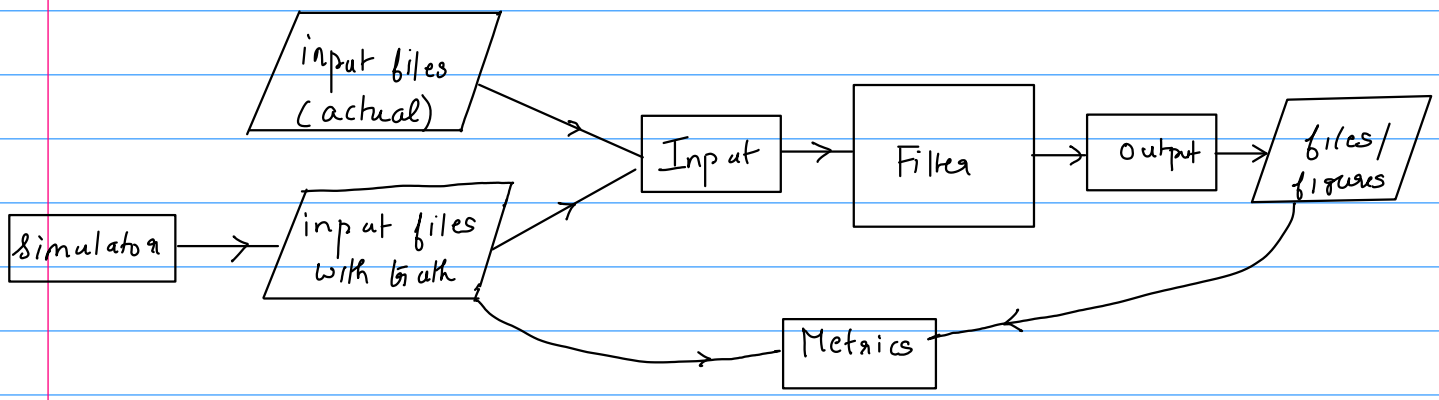
even within a block use simple functions for implementing the procedure

For simulated data it would be possible to actually see the performance of our algorithm, (because we know the ground truth). So we might need another module to do that. This metrics module should compare the actual data - x-s with the filtered yt-s The comparison can be done in terms of sum of squared errors, mean squared error, or sum of absolute errors. How will you design a flexible Metrics module?

```python
import numpy as np

class Metrics:
    def __init__(self, metric_type):
        self.metric_type = metric_type

    def get_ground_truth(self, ground_truthfile):
        pass
        # what to return from this?
    def get_output_data(self, output_file):
        pass
        # what to return from this?

    def compute_metric(self, ground_truthfile, output_file):
        truth = self.get_ground_truth(ground_truthfile)
        data = self.get_output_date(output_file)

        if self.metric_type = "mse":
            self.compute_mse(truth, data)
        elif self.metric_type = "sse":
            pass
        elif self.metric_type = "sae":
            pass

    def compute_mse(self, truth, data):
        pass

# How to test this module?
```

again to have intermediate data available and for flexibility - choose input and output to be files.

Now our data processing system looks like this



Now we need to integrate all of this into a workflow.

```python
from Simulator import *
from Input import *
from Filter import *
from Output import *
from Metrics import *

# Setup the parameters in your workflow
v = 1
delta_t = 0.001
x0 = 0
sigmasq = 1
input_data_file = "inputdata_file_experiment_1.csv"
infile_type = "csv"
output_data_file = "outputdata_file_experiment_1.csv"
outfile_type = "csv"
metrics_type = "mse"
alpha = 0.1

# Setup the components in your workflow or system
sim = Simulator(v, delta_t, x0, sigmasq)
inp = Input(infile_type, input_data_file)
metric_computation = Metrics(metrics_type)
outp = Output(outfile_type, output_data_file)
datafilter = Filter(alpha)

# Generate some test data
sim.run_simulate(input_data_file, number_of_times)
# A workflow which processes the data in blocks
data = inp.get_input()
datafilter.initialize_y(data[0]) # depends on the data type here
datafilter.process_multiple_data_points(data[1:])
outp.produce_output()

# A workflow which processes the data one at a time - good for interactive tuning, debugging etc
```

Other important facts to keep in mind.

- during experimentation and testing treat storage as being cheap. Multiple directories for experiments. Maintain temporary data.
- use version control for your code.

- git init
- git add
- git commit
- git status

} some common commands.

Evaluation.

Each group should take up a module and complete it

- complete all functions
- find and fix errors
- specify inputs and outputs
- comment the code
- test the module by writing test functions with sample inputs.

The modules are Filter.py, Input.py, Output.py, Simulator.py, Metrics.py Experiment_1.py. Maximum marks is 10.

If more than two groups choose a module; say n groups choose ($n > 2$), then the maximum mark for each group will be $\left(\frac{10}{n-1}\right)$.