Interpolation

In this lesson, we will learn about the need for interpolation and how to implement it.

WE'LL COVER THE FOLLOWING

The need for interpolation

If a reading is missing between two consecutive readings, one approach is to replace the missing value using interpolation over the values that are present.

The signal in a weakly linked circuit is a good real-world example of a noisy signal. It is easy to identify the shape of the signal by looking at it, but difficult to quantify its mathematical form. This causes a lot of frustration amongst engineers, and that is why interpolation is a necessary tool to have.

The `interp1d()` function from the `scipy.interpolate` sub-package interpolates the data based on splines of varying order. `interp1d` returns an object which behaves like a function. When this function is given an arbitrary value of x, it returns the corresponding interpolated y value.

This is a very useful technique when plotting data. Let's see a code example of this along with the graphs below:

```
from scipy import *
from scipy.interpolate import *
import matplotlib.pyplot as plt

def f(x):           # function for signal
    return (5 * sin(x)) + (0.1 * x**2) + (0.01 * x**3)

n = 10
x = linspace(-2 * pi, 2 * pi, n)   # generating xdata
noise = randn(n)    # generating random noise
y = f(x) + noise        # adding noise to the signal
```

```
# plotting signal
fig, ax = plt.subplots()
ax.plot(x, y, 's', marker='.')  # the third argument creates a scatter plot
ax.set_title('Signal')
```

The graph above has a noisy signal and does not make much sense. To make the graph look more useful, we will use interpolation. There are different kinds of interpolation that can be implemented in Python, such as linear, quadratic, cubic, and nearest. These can be implemented according to the needs of the user.

> 💡 Use the `help('scipy.interpolate.interp1d')` command to explore more properties.

The third argument of the `interp1d()` function determines the type of interpolation. We declare this by assigning the type to the property `kind`.

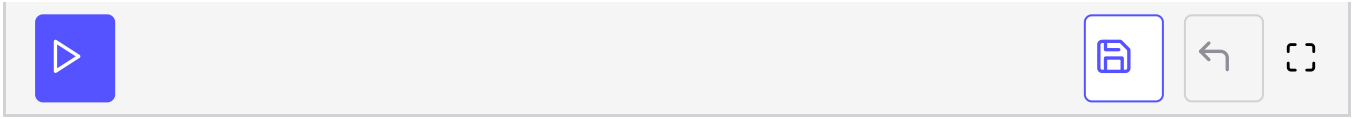
```
interp1d(x, y, kind='type')
```

Implementation

Let's see a useful implementation of the `interp1d` function:

```
from scipy import *
from scipy.interpolate import *
import matplotlib.pyplot as plt

def f(x):           # function for signal
    return (5 * sin(x)) + (0.1 * x**2) + (0.01 * x**3)

n = 10
x = linspace(-2*pi, 2*pi, n)   # generating xdata
noise = randn(n)    # generating random noise
y = f(x) + noise        # adding noise to the signal

x_plotter = linspace(-2*pi, 2*pi, 100)
y0 = interp1d(x, y, kind='linear')    # interpolation of type linear
y1 = interp1d(x, y, kind='quadratic')  # interpolation of type quadratic

# plotting the signal and interpolations
fig, ax = plt.subplots()
```

```
ax.plot(x, y, 's', marker='o', color='k', label='Noisy Data')  # signal
ax.plot(x_plotter, y0(x_plotter), label='linear')
ax.plot(x_plotter, y1(x_plotter), label='quadratic')

ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('Interpolation')

fig.tight_layout()
```



Using a function, we have generated an arbitrary signal in lines 5 and 6. In line 11, we add some noise to the signal to simulate random behavior in the data generated. In line 13, we have generated the data for the x-axis of the interpolation objects `y0` and `y1`.

The range of `x_plotter` is the same as that of `x` except it has more values. With more points to graph, this ensures that our interpolation is smoother and more precise. In lines 14 and 15, we create 2 interpolation objects whose kinds are `linear` and `quadratic` respectively.

In line 20, we plot the original signal `y` for comparison. The linear and quadratic interpolated signals are plotted on lines 21 and 22. Note that the interpolation objects `y0` and `y1` act like a function and plot the data according to the input.

Let’s test your understanding of the concepts learned so far with a quiz.