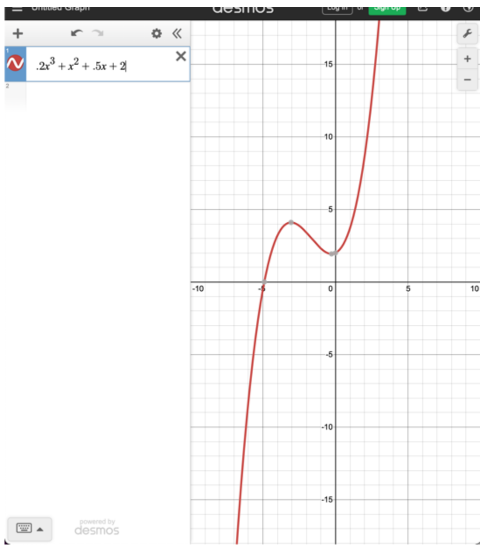
EE 104 – DOCUMENTATION

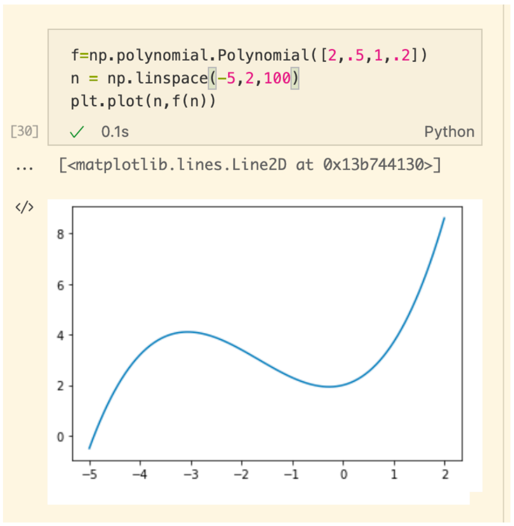
REGRESSION

The polynomial used for many of the regression methods is shown below.

$$y = .2x^3 + x^2 + .5x + 2$$



Fitting this polynomial turned out to be relatively simple. All regression methods seemed to work well. The regression was taken using the window from x (-5,2).

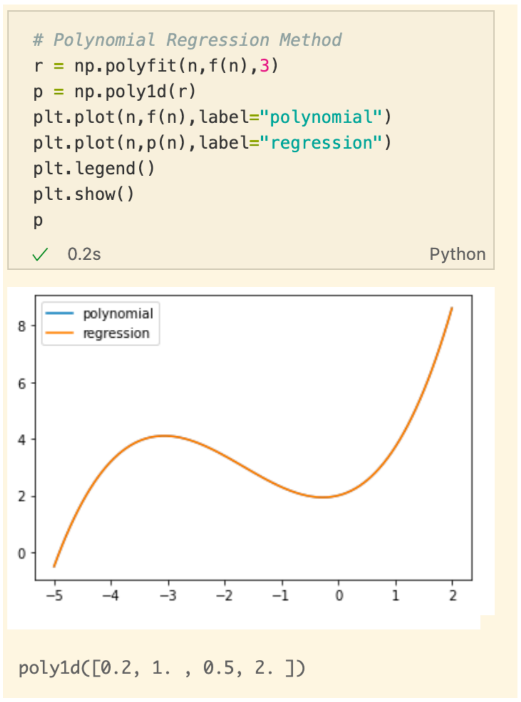


```
f=np.polynomial.Polynomial([2,.5,1,.2])
n = np.linspace(-5,2,100)
plt.plot(n,f(n))
```

```
[<matplotlib.lines.Line2D at 0x13b744130>]
```

Polynomial Regression

Polynomial regression was done using numpy's polyfit tool. Using this package we were able to get a completely perfect fit.



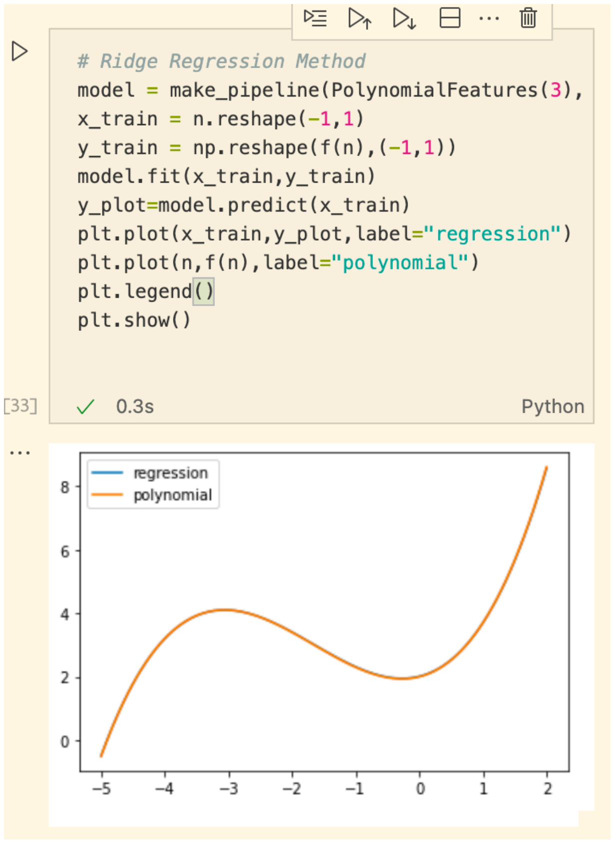
```python
# Polynomial Regression Method
r = np.polyfit(n,f(n),3)
p = np.poly1d(r)
plt.plot(n,f(n),label="polynomial")
plt.plot(n,p(n),label="regression")
plt.legend()
plt.show()
p
```

✓ 0.2s Python

```
poly1d([0.2, 1. , 0.5, 2. ])
```

Ridge Regression

Sklearn’s Ridge regression is used to fit the same polynomial. In this case a pipeline is used to tell the ridge regression method how many polynomial features the data set has. When noting 3 polynomial features the ridge regression also yields a perfect result.

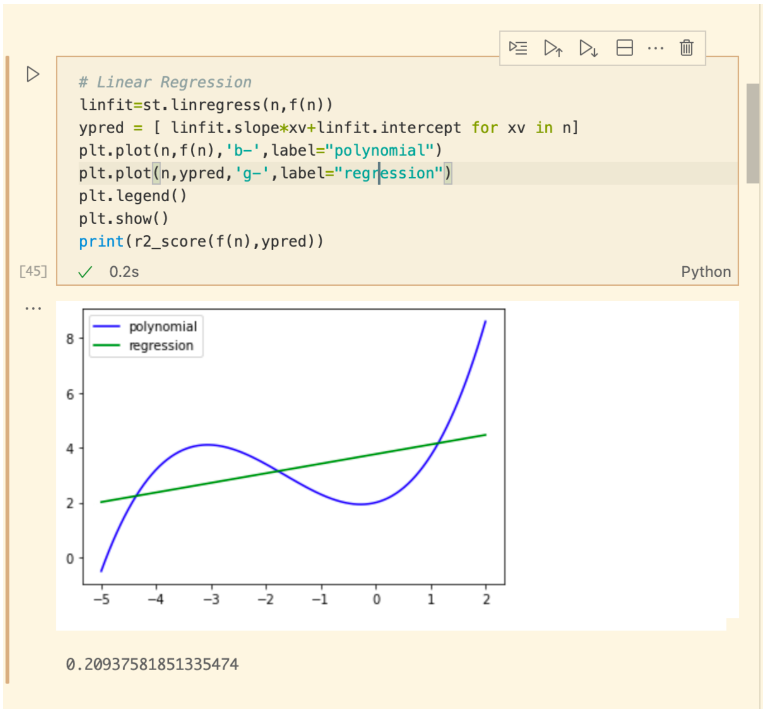


```
# Ridge Regression Method
model = make_pipeline(PolynomialFeatures(3),
x_train = n.reshape(-1,1)
y_train = np.reshape(f(n),(-1,1))
model.fit(x_train,y_train)
y_plot=model.predict(x_train)
plt.plot(x_train,y_plot,label="regression")
plt.plot(n,f(n),label="polynomial")
plt.legend()
plt.show()
```

```
[33]  ✓ 0.3s                                   Python
```

Linear Regression

The linear regression package from sklearn is used to fit this polynomial as well. As expected with linear regression a perfect fit is not possible, however the trend can still be predicted.



```
# Linear Regression
linfit=st.linregress(n,f(n))
ypred = [ linfit.slope*xv+linfit.intercept for xv in n]
plt.plot(n,f(n),'b-',label="polynomial")
plt.plot(n,ypred,'g-',label="regression")
plt.legend()
plt.show()
print(r2_score(f(n),ypred))
```

[45] ✓ 0.2s Python

```
0.20937581851335474
```

As can be seen from the figure above the general trend is correct however the coefficient of determination R^2 is very low.

Damped Sine Wave

An arbitrary damped sine wave is fit as well. The sine wave used is shown below. The window is limited to x: (-2,3) and y(-2,2).

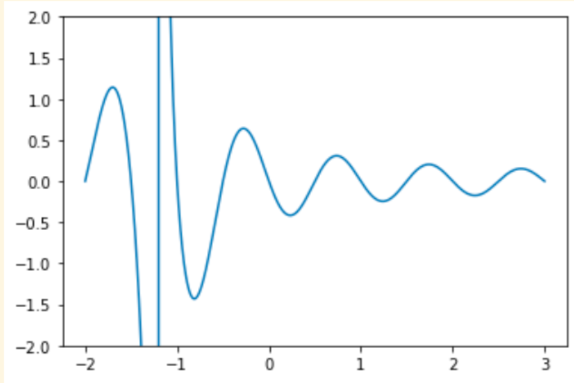
damped sine wave

$$\left(\frac{a}{b+x}\right)\sin\left(\left(\frac{2\pi}{c}\right)x\right)+d$$

```python
a=0.6
b=1.2
c=-1
d=0
x = np.linspace(-2,3,10000)
f = lambda x: a/(b+x)*np.sin((2*np.pi/c)*x)+d
plt.ylim(-2,2)
plt.plot(x,f(x))
```

✓ 0.1s Python

```
[<matplotlib.lines.Line2D at 0x13b75f250>]
```



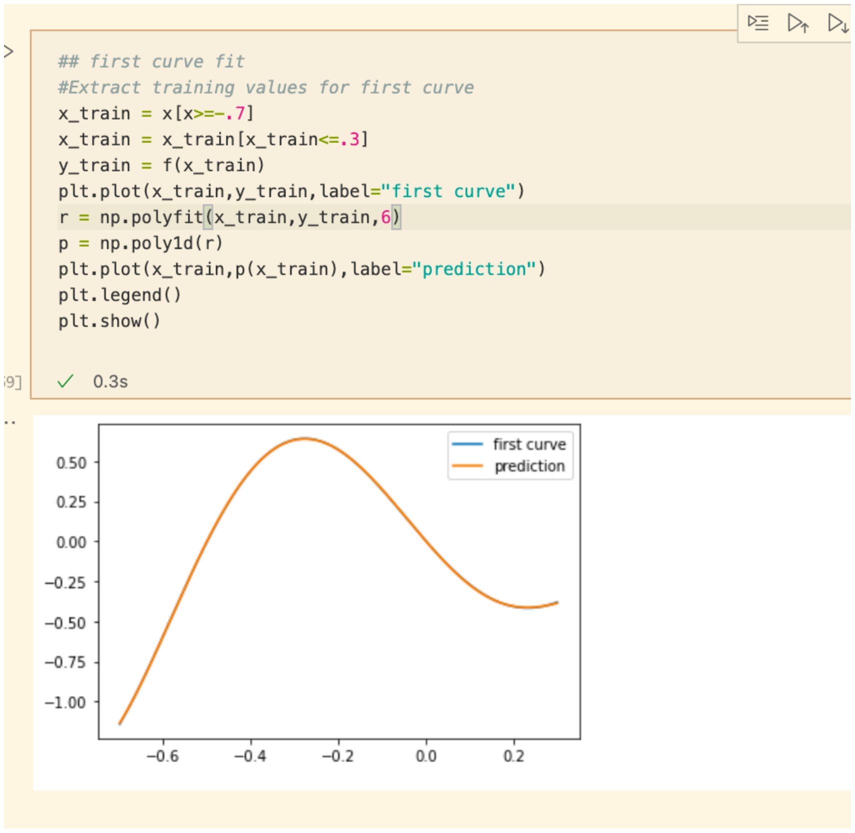
Since fitting the entire function is not possible using the previously mentioned regression methods, we will use polynomial regression to fit the first curve of the sine wave using limits

x: (-.7,.3).

Using a polynomial regression with a high enough order can yield a perfect fit for any individual curve. It is however impossible to fit the entire function since we do not have a method to fit continuously oscillating functions outside of the fitting window.

First Curve Fit

We are able to perfectly fit this curve with a high enough polynomial regression. It can be expected that the regression fails outside this window



```
## first curve fit
#Extract training values for first curve
x_train = x[x>=-.7]
x_train = x_train[x_train<=.3]
y_train = f(x_train)
plt.plot(x_train,y_train,label="first curve")
r = np.polyfit(x_train,y_train,6)
p = np.poly1d(r)
plt.plot(x_train,p(x_train),label="prediction")
plt.legend()
plt.show()
```

✓ 0.3s

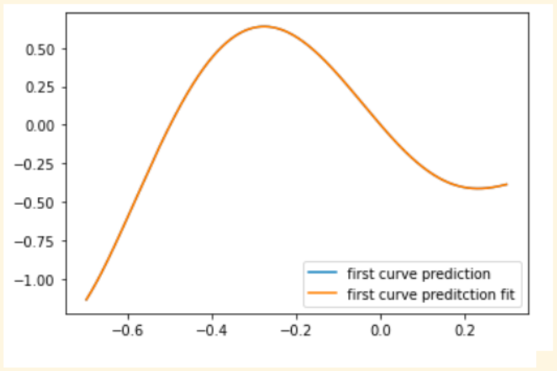
Fitting the First Curve Fit

Fitting the previous curve fit also yields perfect results.

```
## Fitting predicted curve
r2 = np.polyfit(x_train,p(x_train),6)
p2 = np.poly1d(r2)
plt.plot(x_train,p(x_train),label="first
plt.plot(x_train,p2(x_train),label="first
plt.legend()
p2
```

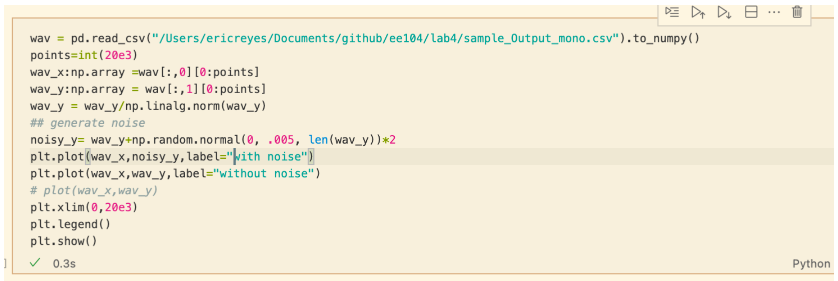
```
[62]  ✓ 0.3s                                   Python
```

```
...  poly1d([-1.08938656e+01, -3.83716003e+01,
     -1.23824800e+01,  1.98504406e+01,
             2.56406076e+00, -3.17405319e+00,
     1.66492172e-04])
```



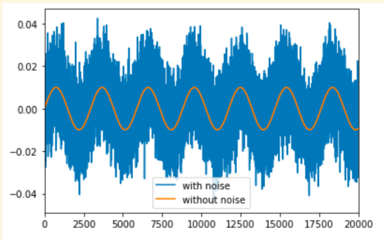
SINUSOIDAL WITH NOISE FITTING

Pandas is used to retrieve data from a csv file for an arbitrary sinusoidal tone. The original tone is normalized and plotted below.

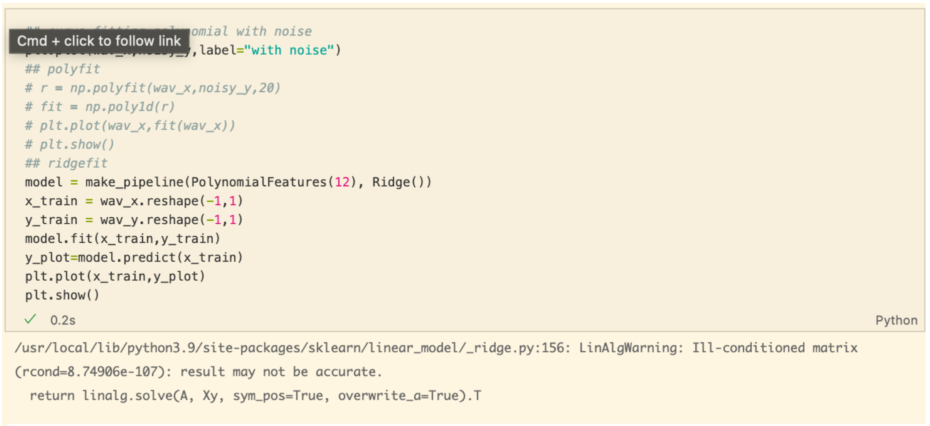


```
wav = pd.read_csv("/Users/ericreyes/Documents/github/ee104/lab4/sample_Output_mono.csv").to_numpy()
points=int(20e3)
wav_x:np.array =wav[:,0][0:points]
wav_y:np.array = wav[:,1][0:points]
wav_y = wav_y/np.linalg.norm(wav_y)
## generate noise
noisy_y= wav_y+np.random.normal(0, .005, len(wav_y))*2
plt.plot(wav_x,noisy_y,label="with noise")
plt.plot(wav_x,wav_y,label="without noise")
# plot(wav_x,wav_y)
plt.xlim(0,20e3)
plt.legend()
plt.show()
```

✓ 0.3s Python



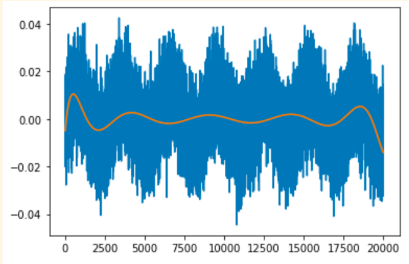
Noise is added to this tone, which is then curve fit using ridge regression. In the figure below there is too much noise for the fit to work perfectly, but it does seem to represent where the wave crosses the x-axis correctly.



```
## curve fitting polynomial with noise
plt.plot(wav_x,noisy_y,label="with noise")
## polyfit
# r = np.polyfit(wav_x,noisy_y,20)
# fit = np.poly1d(r)
# plt.plot(wav_x,fit(wav_x))
# plt.show()
## ridgefit
model = make_pipeline(PolynomialFeatures(12), Ridge())
x_train = wav_x.reshape(-1,1)
y_train = wav_y.reshape(-1,1)
model.fit(x_train,y_train)
y_plot=model.predict(x_train)
plt.plot(x_train,y_plot)
plt.show()
```

✓ 0.2s Python

```
/usr/local/lib/python3.9/site-packages/sklearn/linear_model/_ridge.py:156: LinAlgWarning: Ill-conditioned matrix
(rcond=8.74906e-107): result may not be accurate.
  return linalg.solve(A, Xy, sym_pos=True, overwrite_a=True).T
```



Multivariable Regression

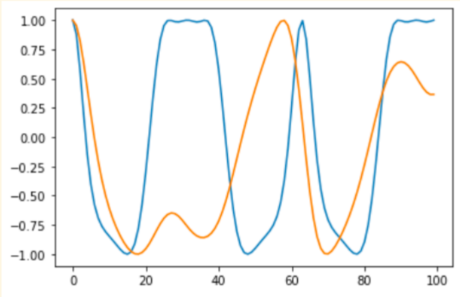
Sklearn's optimize package contains a curve fitting tool that is able to guess the inputs to a multivariable function. An arbitrary multivariable function is created and the curve fit tool is tested with different guesses.

```
## multivariate variable regression

x0 = [x/10 for x in range(100)]
x1 = [np.sin(x/10) for x in range(100)]
tupx=(x0,x1)
rf = lambda tupx,fx0,fx1: np.cos(np.multiply(tupx[0],fx0)+np.sin(np.multiply(tupx[1],fx1)))
y = rf(tupx,2,3)
plt.plot(y)
popt,pcov=opt.curve_fit(rf,tupx,y) # no guess
plt.plot(rf(tupx,*popt))
```

1] ✓ 0.1s

```
[<matplotlib.lines.Line2D at 0x13bc80fd0>]
```

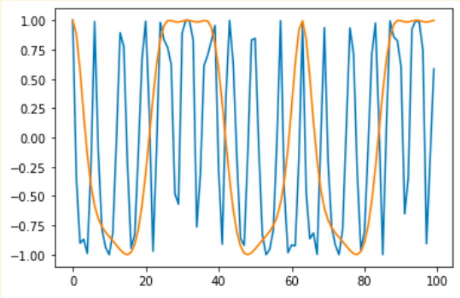


With no guess the curve fit is unable to correctly guess the function paramaters, but seems too be relatively close

```
popt,pcov=opt.curve_fit(rf,tupx,y,(10,10)) # too large guess
plt.plot(rf(tupx,*popt))
plt.plot(y)
```

2] ✓ 0.1s

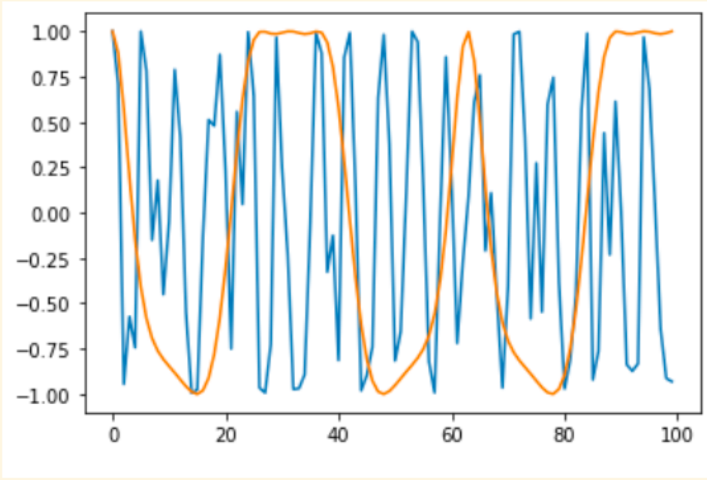
```
[<matplotlib.lines.Line2D at 0x13ba004c0>]
```



```
popt,pcov=opt.curve_fit(rf,tupx,y,(0,0)) # guessing 0
plt.plot(rf(tupx,*popt))
plt.plot(y)
```

✓ 0.1s

```
[<matplotlib.lines.Line2D at 0x13bab13a0>]
```

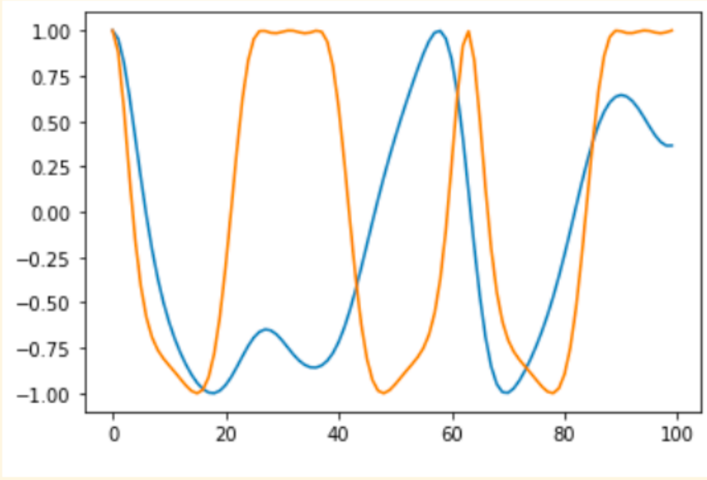


Interestingly, guessing (0,0) yields the same result as too large of a guess

```
popt,pcov=opt.curve_fit(rf,tupx,y,(1,1)) # small guess
plt.plot(rf(tupx,*popt))
plt.plot(y)
```

✓ 0.1s

```
[<matplotlib.lines.Line2D at 0x13b78e2e0>]
```



1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
20
40
60
80
100