Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel$\rightarrow$Restart) and then **run all cells** (in the menubar, select Cell$\rightarrow$Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [174]:

```
NAME = "Nutapol Thungpao"
ID = "122148"
```

Lab 02: Nonlinear Regression and Overfitting

In Lab 01, we explored the construction of linear regression models. Recall the assumptions we make in linear regression:

- $\textbf{x} \in {\cal X} = \mathbb{R}^n$
- $y \in {\cal Y} = \mathbb{R}$
- The $\textbf{x}$ data are drawn i.i.d. from some (unknown) distribution over ${\cal X}$
- There is a linear relationship between $\textbf{x}$ and y with additive constant-variance Gaussian noise, i.e., $y \sim {\cal N}(\theta^\top \textbf{x}, \sigma^2)$, where $\theta \in \mathbb{R}^{n+1}$ is unknown and $\textbf{x}$ is an $n+1$-dimensional vector augemented with a constant value of 1 as its first element.

Today, we consider what we might do when the fourth assumption, linearity, does not hold. We introduce a particular form of nonlinear regression, *polynomial regression*, in which we account for nonlinear relationships between $\mathbf{x}$ and y by performing nonlinear transformations of the input variables in $\mathbf{x}$.

As an example, if we had a single input variable x, linear regression gives us the hypothesis $$h_\theta(x) = \theta_0 + \theta_1 x .$$ We can add a new "variable" x^2, which is a nonlinear transformation of the input x: $$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 .$$ The important thing to notice here is that although the hypothesis is *nonlinear* in x, allowing us to model a more complex function than ordinary linear regression, the hypothesis is *linear* in θ, allowing us to use the normal equations to find the optimal θ as before.

Polynomial Regression

More generally, polynomial regession is a form of linear regression in which the relationship between the independent variables $\mathbf{x}$ and the dependent variable y is modelled as a polynomial.

For a single input x, the hypothesis in a polynomial regression of degree d is $$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_d x^d$$ $$h_\theta(x) = \sum_{i=0}^{d} \theta_i x^i$$

For a multivariate input $\mathbf{x}$, we introduce terms corresponding to every degree-d combination of factors. For example, if $n=3$ and $d=2$, we have $$h_\theta(\mathbf{x}) = \theta_0

```
+ \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3
+ \theta_4 x_1^2 + \theta_5 x_1 x_2 + \theta_6 x_1 x_3
+ \theta_7 x_2^2 + \theta_8 x_2 x_3 + \theta_9 x_3^2 .$$
```

Example 1: Synthetic data with a quadratic nonlinearity

Let's take a look at how polynomial regression as compared to simple linear regression model works for data with a simple quadratic nonlinearity.

Generate a synthetic dataset

First, we generate 100 observations from a ground truth quadratic function with Gaussian noise:

In [175]:

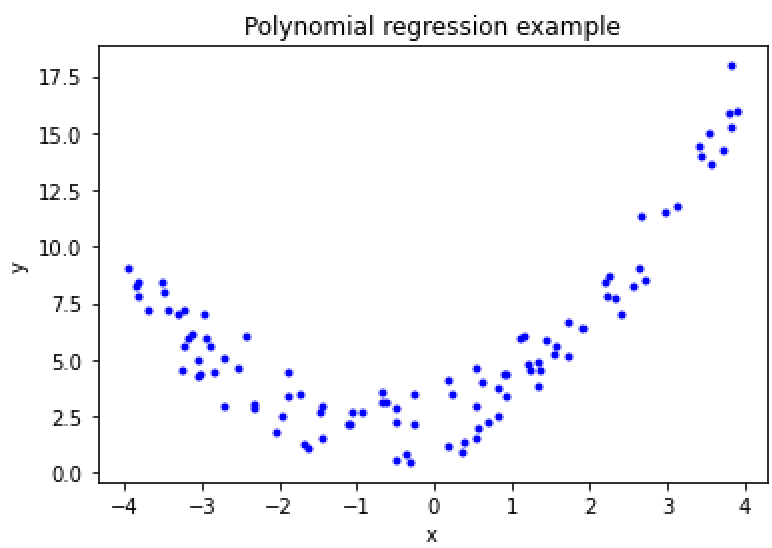
```
import matplotlib.pyplot as plt
import numpy as np
import random

# please do not change the random seed, or the autograder's result checking will be
wrong!

np.random.seed(0)
random.seed(0)
```

In [176]:

```
# Generate X
m = 100
X = np.random.uniform(-4, 4, (m,1))

# Generate y
a = 0.7
b = 1
c = 2
y = a * X**2 + b * X + c + np.random.randn(m, 1)
```

In [177]:

```
# Plot
plt.plot(X, y, 'b.')
plt.title('Polynomial regression example')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Implement the hypothesis function

First, we will use ordinary linear regression: $$h_\theta(x) = \theta_0 + \theta_1 x$$ Then, we use polynomial regression with $d=2$: $$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 $$ In either case, by letting the input vector $\bf{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$ or $\bf{x} = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}$ appropriately, the hypothesis can be written $$ h_\mathbf{\theta}(\mathbf{x}) = \mathbf{\theta}^\top \mathbf{x} . $$

Let's implement this hypothesis function in Python:

In [178]:

```
def h(X, theta):
    return X.dot(theta)
```

Implement the regression function (normal equations)

Recall the normal equations used to find the θ minimizing $J(\theta)$ when the design matrix $\mathtt{X}_{m\times(n+1)}$ contains one row for each example and $\mathbf{y}$ is a column vector: $$\mathbf{\theta} = (\mathtt{X}^\top \mathtt{X})^{-1}\mathtt{X}^\top\mathbf{y}$$

Let's implement the normal equations in Python:

In [179]:

```
def regression(X, y):
    cov = np.dot(X.T, X)
    cov_inv = np.linalg.inv(cov)
    theta = np.dot(cov_inv, np.dot(X.T, y))
    return theta
```

Exercise 1.1 (2 points)

Create a Python function to calculate the RMSE (root mean squared error) for a set of predictions $\hat{\mathbf{y}}$: $$\mathrm{RMSE}(\mathbf{y},\hat{\mathbf{y}}) = \sqrt{\frac{\sum_{i=1}^{m} \left(y^{(i)}-\hat{y}^{(i)} \right)^2} {m}}$$

In [180]:

```
def rmse(y, y_pred):
    # YOUR CODE HERE
    sum_y = np.sum((y-y_pred)**2)
    error = np.sqrt(sum_y/np.shape(y)[0])
    #raise NotImplementedError()
    return error
```

In [181]:

```
print(rmse(np.array([1,1.1,2,-1]), np.array([1.1,1.3,1.5,0.1])))

# Test function: Do not remove
assert np.round(rmse(np.array([1,1.1,2,-0.1]), np.array([1.1,1.3,1.5,0.1])), 5) ==
np.round(0.291547594742265O5, 5), "calculate rmse incorrect"
print("success!")
# End Test function
```

```
0.6144102863722254
success!
```

Expected output: 0.6144102863722254

Implement a simple linear model

OK, as stated earlier, let's implement a simple linear model:

In [182]:

```
# Add intercept column of all 1's
X_aug = np.insert(X, 0, 1, axis=1)

# Print first 5 rows of X
print(X_aug[0:5,:])

# Find optimal parameters
theta_slr = regression(X_aug, y)

# Predict y
y_pred_slr = h(X_aug, theta_slr)

print('Linear regression RMSE: %f' % rmse(y, y_pred_slr))
```

```
[[ 1.          0.39050803]
 [ 1.          1.72151493]
 [ 1.          0.82210701]
 [ 1.          0.35906546]
 [ 1.         -0.61076161]]
Linear regression RMSE: 3.413803
```

Exercise 1.2 (2 points)

From the simple linear model above, create another linear model using a **polynomial** model with degree $d=2$. You need to implement these steps:

- Create the design matrix $\mathtt{X}$ as numpy matrix `X_aug` similarly to how we set up `X_aug` above.
- Find the optimal solution θ as numpy vector `theta_pr` similarly to how we set up `theta_slr` above.

Hint here!

In [183]:

```
# Add intercept column of all 1's
X_squer=X**2
#print(np.shape(X_aug))
X_aug = np.insert(X_aug, 2, X_squer.T, axis=1)
# Print first 5 rows of X
#print(X_aug[0:5,:])

# Find optimal parameters
theta_pr = regression(X_aug, y)
```

In [184]:

```
# Predict y
y_pred_pr = h(X_aug, theta_pr)
print(X_aug[0:5,:])
print('Polynomial regression RMSE: %f' % rmse(y, y_pred_pr))

# Test function: Do not remove
assert np.array_equal(np.round(theta_pr.T), np.round([[1.90932595, 1.02311816, 0.71747835]])), "theta_pr are incorrect"
assert np.round(X_aug[10,1] ** 2, 5) == np.round(X_aug[10,2], 5), "X_aug are incorrect"
assert np.round(rmse(y, y_pred_pr) ** 2 * y.shape[0], 5) == np.round(np.dot((y - y_pred_pr).T, y - y_pred_pr), 5), "RMSE incorrect"
print("success!")
# End Test function
```

```
[[ 1.          0.39050803  0.15249652]
 [ 1.          1.72151493  2.96361366]
 [ 1.          0.82210701  0.67585993]
 [ 1.          0.35906546  0.12892801]
 [ 1.         -0.61076161  0.37302974]]
Polynomial regression RMSE: 0.986690
success!
```

Expected output \ [[1. 0.39050803 0.15249652]\ [1. 1.72151493 2.96361366]\ [1. 0.82210701 0.67585993]\ [1. 0.35906546 0.12892801]\ [1. -0.61076161 0.37302974]]\ Polynomial regression RMSE: 0.986690

Compare the two different models using RMSE

We see that the degree 2 polynomial fit is much better, reducing average error from 3.4 to 0.99.

To further visualize the performance of our model, we should plot the predictions vs. the observed data.

Exercise 1.3 (2 points)

This one is a bit tricky.

We'd like to write a function `get_predictions` that works for any model degree depending on what θ it is passed. The function should take as input a vector of scalar x values along with a set of parameters θ. It should output a vector of predictions $\hat{\mathbf{y}}$.

Your `get_predictions` function needs to construct an appropriate design matrix $\mathtt{X}$ then use the hypothesis function we already wrote earlier.

Hint here!

In [185]:

```
def get_predictions(x, theta):
    # Change the shape of x to support the function
    x = np.array([x]).T
    X_aug = np.insert(x, 0, 1, axis=1)
    while X_aug.shape[1] < theta.shape[0]:
        newcol = X_aug[:,-1:]*x
        X_aug=np.insert(X_aug,X_aug.shape[1],newcol.T,axis=1)

    y_hat = X_aug@theta
    return y_hat
```

In [186]:

```
x_series = np.linspace(-4, 4, 1000)
y_series_slr = get_predictions(x_series, theta_slr)
y_series_pr = get_predictions(x_series, theta_pr)

print("y_series_slr:", y_series_slr[2:5].T)
print("y_series_pr:", y_series_pr[2:5].T)

# Test function: Do not remove
assert np.round(get_predictions(np.array([1, 9, 2, -9]), theta_slr).T, 5) is not No
ne, "predict from theta_slr is incorrect"
assert np.round(get_predictions(np.array([1, 1, 0.1, 2]), theta_pr).T, 5) is not No
ne, "predict from theta_pr is incorrect"
print("success!")
# End Test function
```

```
y_series_slr: [[2.72462183 2.73101513 2.73740842]]
y_series_pr: [[9.0812643  9.04632656 9.01147497]]
success!
```

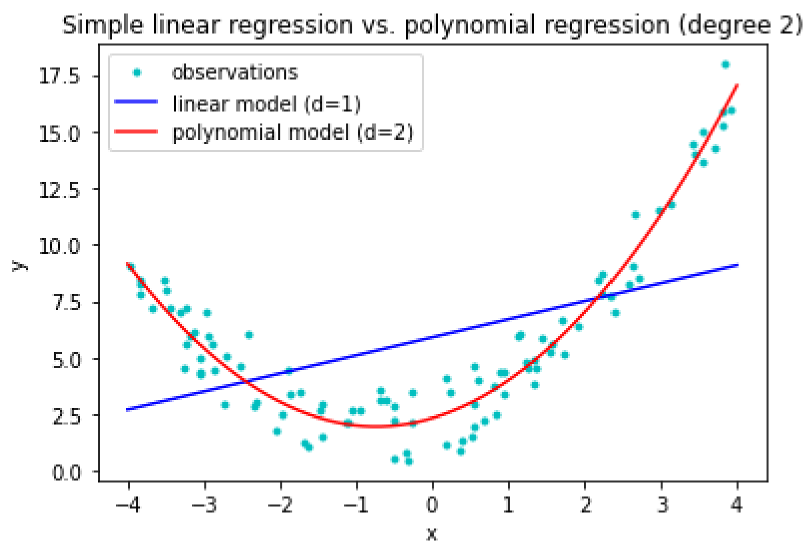
Expected output:\ y_series_slr: [[2.72462183 2.73101513 2.73740842]]\ y_series_pr: [[9.0812643 9.04632656 9.01147497]]

Plot x against y with the two regression models

Now that we have a working `get_predictions()`, we can plot the data with the linear and quadratic model:

In [187]:

```
plt.plot(X[:,0], y, 'c.', label='observations')
plt.plot(x_series, y_series_slr, 'b-', label='linear model (d=1)')
plt.plot(x_series, y_series_pr, 'r-', label='polynomial model (d=2)')
plt.title('Simple linear regression vs. polynomial regression (degree 2)')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



From the plot, we see clearly that the quadratic model is a much better fit to the data.

Compare models using goodness of fit

Besides RMSE, let's also get the R^2 for our two models. Recall the formula for R^2: \begin{align} \ R^2 = 1 - \frac{\sum_{i=1}^{m} \left(y^{\left(i\right)}-\hat{y}^\left(i\right) \right)^2} {\sum_{i=1}^{m} \left(y^{\left(i\right)}-\bar{y}^\left(i\right) \right)^2} \end{align}

Exercise 1.4 (2 points)

Fill in the function `r_squared()` using the equation above.

Hint here!

In [188]:

```
def r_squared(y, y_pred):
    # YOUR CODE HERE
    up=np.sum((y-y_pred)**2)
    low = np.sum((y-np.mean(y))**2)
    r_sqr = 1-(up/low)
    #raise NotImplementedError()
    return r_sqr
```

In [189]:

```
print('Fit of simple linear regression model: %.4f' % r_squared(y, y_pred_slr))
print('Fit of polynomial regression model: %.4f' % r_squared(y, y_pred_pr))

# Test function: Do not remove
assert np.round(r_squared(np.array([1, 2, 3]), np.array([1, 2, 3]))) == np.round(1.
0), "r_squared is incorrect"
assert np.round(r_squared(y, y_pred_pr), 4) == np.round(0.9353, 4), "r_squared is i
ncorrect"
print("success!")
# End Test function
```

```
Fit of simple linear regression model: 0.2254
Fit of polynomial regression model: 0.9353
success!
```

Expected output:\ Fit of simple linear regression model: 0.2254\ Fit of polynomial regression model: 0.9353

So we see again the superior fit of the quadratic model using R^2 (0.94 vs. 0.23).

Compare models using residual histograms

Next, let's look at another useful analysis: histograms of each model's residuals. Rather than summarizing the residuals with RMSE or R^2, we'll need a function to calculate a vector of residuals. Then we'll be able to make histograms.

Exercise 1.5 (2 points)

Fill in function `residual_error()` to find the residual error vector $\mathbf{y} - \hat{\mathbf{y}}$.

Once we have that function, we can calculate `error_slr` for the simple linear regression and `error_pr` for the polynomial regression.

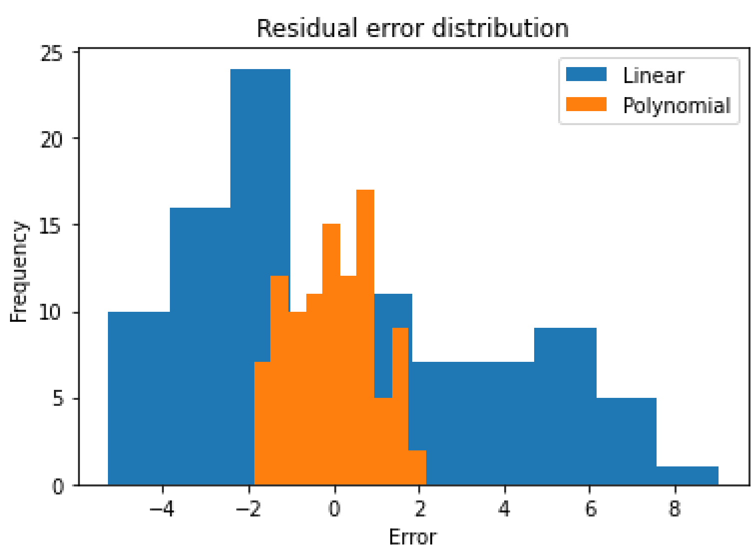
In [190]:

```
def residual_error(y, y_pred):
    # YOUR CODE HERE
    error=y-y_pred
    #raise NotImplementedError()
    return error

error_slr = residual_error(y, y_pred_slr)
error_pr = residual_error(y, y_pred_pr)
```

In [191]:

```
# Plot distribution of residual error for each model
print("error_slr sample:", error_slr[0:5, 0].T)
print("error_pr sample:", error_pr[0:5, 0].T)

plt.hist(error_slr, bins=10, label = 'Linear')
plt.hist(error_pr, bins=10, label = 'Polynomial')
plt.xlabel('Error')
plt.ylabel('Frequency')
plt.title('Residual error distribution')
plt.legend()
plt.show()

# Test function: Do not remove
assert np.array_equal(np.round(get_predictions(np.array([1, 9, 2, -9]), theta_slr).
T),
                      np.round([[6.70364883, 13.09055058, 7.50201155, -1.27997835
]])), "predict from theta_slr is incorrect"
assert np.array_equal(np.round(get_predictions(np.array([0, 7, 1.5, -0.3]), theta_p
r).T),
                      np.round([[2.34050076, 42.14663283, 5.3284002, 2.10566904
]])), "predict from theta_pr is incorrect"
print("success!")
# End Test function
```

```
error_slr sample: [-4.88494741 -0.58280848 -2.8007543  -5.27887921 -2.2
7906541]
error_pr sample: [-1.49521216  0.67105966  0.15715854 -1.86746535  1.14
869785]
```




```
success!
```

Expected output:\ error_slr sample: [-4.88494741 -0.58280848 -2.8007543 -5.27887921 -2.27906541]\ error_pr sample: [-1.49521216 0.67105966 0.15715854 -1.86746535 1.14869785]

The residual plot again shows clearly how much better the polynomial model is than the linear model.

Example 2: Sales data

Next, let's model some real data, in particular, monthly sales data from Kaggle using polynomial regression with varying degree.

We will observe the effects of varying the degree of the polynomial regression fit on the prediction accuracy.

However, as discussed in class, as models become more complex, we will encounter the issue of *overfitting*, in which a too-powerful model starts to model the noise in the specific training set rather than the overall trend.

To ensure that we're not fitting the noise in the training set, we will split the data into seaparte train and test/validation datasets. The training dataset will consist of 60% of the original observations, and the test dataset will consist of the remaining 40% of the observations.

For various polynomial degrees, we'll estimate optimal parameters θ, from the training set, then we'll use the test/validation dataset to measure the accuracy of the optimized model.

First, let's read the data from the CSV file and set up variables `X_data` , `y_data` .

In [192]:

```
# Import CSV
data = np.genfromtxt('MonthlySales_data.csv',delimiter = ',', dtype=str)

# Extract headers
headers = data[0,:]
print("Headers:", headers)

# Extract raw data
data = np.array(data[1:,:], dtype=float);
mean = np.mean(data,axis=0)
std = np.std(data,axis=0)
data_norm = (data-mean)/std

# Extract y column from raw data
y_index = np.where(headers == 'sale amount')[0][0];
y_data = data[:,y_index];

# Extract x column (just the month) from raw data
month_index = np.where(headers == 'month')[0][0]
# print(year_index, month_index)
X_data = data[:,[month_index]];
m = X_data.shape[0]
n = X_data.shape[1]
X_data = X_data.reshape(m, n)

print('Extracted %d monthly sales records' % m)
print(X_data.shape)
print(y_data.shape)
```

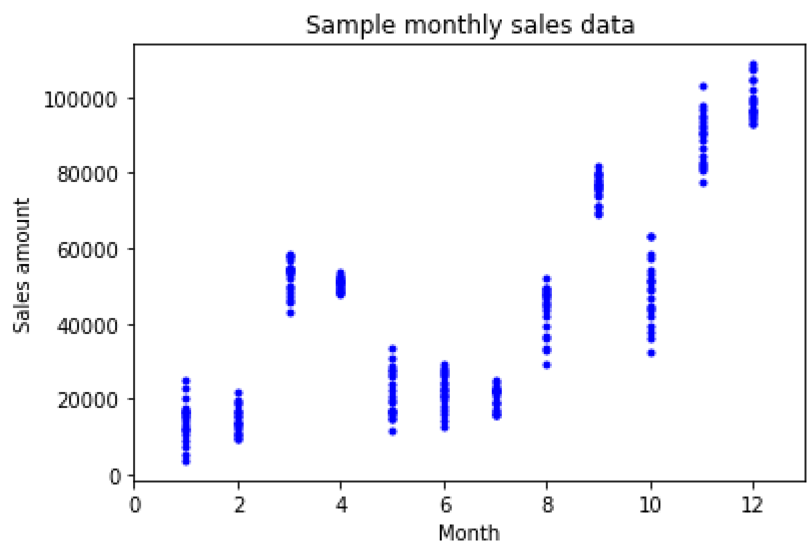
```
Headers: ['year' 'month' 'sale amount']
Extracted 240 monthly sales records
(240, 1)
(240,)
```

Plot the data

Although year and month are discrete variables, they are also ordinal, so they can be treated as real values. Let's plot sales month against sales amount as a scatter plot, and we'll see the discrete nature of the data:

In [193]:

```
fig = plt.figure()
xx1 = X_data[:,0]
yy1 = y_data

plt.plot(xx1, yy1, 'b.')
plt.xlim(0, 13)
plt.xlabel('Month')
plt.ylabel('Sales amount')
plt.title('Sample monthly sales data')
plt.show()
```



Partition the data

Next let's split the overall dataset into subsets for training and validation (test).

Exercise 1.6 (2 points)

Partition `X_data` and `y_data` into training and test datasets

- Let the training set be 60% of the dataset
- Let the rest be the test set
- Shuffle the dataset before splitting it to ensure a similar distribution in the two subsets

You can use the [`random.shuffle()` function (https://www.w3schools.com/python/ref_random_shuffle.asp)](https://www.w3schools.com/python/ref_random_shuffle.asp) </link> to shuffle the indices of the dataset.

In [194]:

```
percent_train = .6

def partition(X, y, percent_train):
    # Create a list of indices into X and y
    idx = np.arange(0,y.shape[0])
    random.seed(1412)
    # just make sure the shuffle always the same please do not remove
    # On your own, do the following:
    # 1. shuffle the idx list
    # 2. Create lists of indices train_idx and test_idx for the train and test sets
    # 3. Set variables X_train, y_train, X_test, and y_test using those index lists

    # YOUR CODE HERE
    random.shuffle(idx)
    size_train=int(y.shape[0]*percent_train)
    train_idx=idx[0:size_train]
    test_idx=idx[size_train:y.shape[0]+1]

    X_train = data[train_idx,1:y_index]
    X_test = data[test_idx,1:y_index]

    y_train = data[train_idx,y_index]
    y_test = data[test_idx,y_index]

    return idx, X_train, y_train, X_test, y_test
```

In [195]:

```
idx, X_train, y_train, X_test, y_test = partition(X_data, y_data, percent_train)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
print(idx[5:9])

# Test function: Do not remove
assert not np.array_equal(np.round(X_data[0:144, :], 3), np.round(X_train,3)), "X_t
rain must be shuffled!"
assert not np.array_equal(np.round(X_data[144:, :], 3), np.round(X_test,3)), "X_tes
t must be shuffled!"
assert not np.array_equal(np.round(y_data[0:144], 3), np.round(y_train,3)), "y_trai
n must be shuffled!"
assert not np.array_equal(np.round(y_data[144:], 3), np.round(y_test,3)), "y_test m
ust be shuffled!"
assert np.array_equal(idx[5:9], [26, 75, 51, 162])
print("success!")
# End Test function
```

```
(144, 1)
(144,)
(96, 1)
(96,)
[ 26  75  51 162]
success!
```

Expected output:\ (144, 1)\ (144,)\ (96, 1)\ (96,)\ [26 75 51 162]

Set up for polynomial regression

Next, let's implement the transformation of a variable x into the expanded list $\begin{bmatrix} x & x^2 & \cdots & x^d \end{bmatrix}$.

Exercise 1.7 (2 points)

Fill in function `x_polynomial()` with code to output a row vector consisting of the elements $x, x^2, \ldots, x^d$, where when d is the degree of the polynomial.

In [196]:

```
def x_polynomial(x, d):
    # YOUR CODE HERE
    X = np.array([x]).T
    X = np.insert(x, 0, 1, axis=1)
    while X.shape[1] < d+1:
        newcol = X[:,-1:]*x
        X=np.insert(X,X.shape[1],newcol.T,axis=1)
    return X
```

In [197]:

```
print(x_polynomial(np.array([[3],[2]]), 5))
print(x_polynomial(np.array([[3],[2]]), 5).shape)

Xi_train = x_polynomial(X_train, 1)
Xi_test = x_polynomial(X_test, 1)

# Test function: Do not remove
assert x_polynomial(np.array([[2],[3]]), 5).shape[1] == 5 + 1, "Size of polynomial
 incorrect"
assert np.array_equal(np.round(x_polynomial(np.array([[2],[3]]), 5), 3),
                      np.round([[1, 2, 4, 8, 16, 32], [1, 3, 9, 27, 81, 243]],3)),
"Polynomial are wrong."
print("success!")
# End Test function
```

```
[[  1   3   9  27  81 243]
 [  1   2   4   8  16  32]]
(2, 6)
success!
```

Expected output:\ [[1. 3. 9. 27. 81. 243.]\ [1. 2. 4. 8. 16. 32.]]\ (2, 6)

Write the cost function

Next let's implmeent to cost function for a given set of parameters θ.

Exercise 1.8 (2 points)

Fill in function `cost()` with appropriate code. Use a constant of $\frac{1}{2m}$ out front.

In [198]:

```
def cost(theta, X, y):
    # YOUR CODE HERE
    sum_function= y-(np.dot(theta,X.T))
    J = (1/(2*y.shape[0]))*np.sum(sum_function**2)
    return J
```

In [199]:

```
# calculate theta
theta = regression(Xi_train, y_train)

# calculate cost in train
J_train = cost(theta, Xi_train, y_train)

y_pred_test = h(Xi_test, theta)
J_test = cost(theta, Xi_test, y_test)

print("J_train:", J_train)
print("J_test:", J_test)

# Test function: Do not remove
assert type(J_train) == np.float64, "Cost function size must be 1"
assert np.round(J_train, 3) == np.round(174395635.44334993, 3), "Cost function for
 train set is wrong"
assert np.round(J_test, 3) == np.round(196382485.91395777, 3), "Cost function for t
est set is wrong"
print("success!")
# End Test function
```

```
J_train: 174395635.44334996
J_test: 196382485.91395798
success!
```

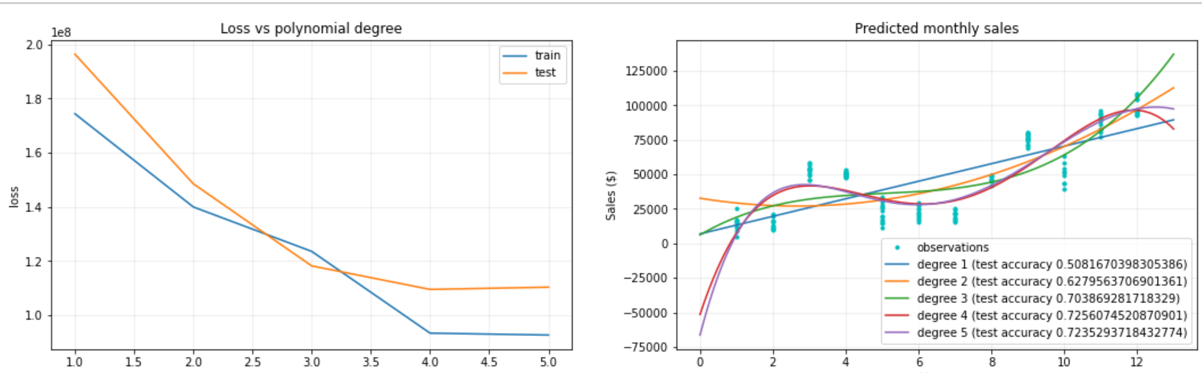
Expected output:\ J_train: 174395635.44334993\ J_test: 196382485.91395777

Try models of varying degree

Next we'll build multiple polynomial regression models with different degree, using sales month as the independent variable and sales amount as the dependent variable.

In [200]:

```
max_degree = 5

J_train = np.zeros(max_degree)
J_test = np.zeros(max_degree)

# Initalize plots for predictions and loss
fig, ax = plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(20)
fig.subplots_adjust(left=.2, bottom=None, right=None, top=None, wspace=.2, hspace=.
2)
plt1 = plt.subplot(1,2,1)
plt2 = plt.subplot(1,2,2)
plt2.plot(X_train, y_train, 'c.', label='observations')

for i in range(1, max_degree+1):
    # Fit model on training data and get cost for training and test data
    Xi_train = x_polynomial(X_train, i)
    Xi_test = x_polynomial(X_test, i);
    theta = regression(Xi_train, y_train)
    J_train[i-1] = cost(theta, Xi_train, y_train)
    y_pred_test = h(Xi_test, theta)
    J_test[i-1] = cost(theta, Xi_test, y_test)

    # Plot
    x_series = np.linspace(0, 13, 1000)
    y_series = get_predictions(x_series, theta)
    plt2.plot(x_series, y_series, '-', label='degree ' + str(i) + ' (test accuracy
 ' + str(r_squared(y_test, y_pred_test)) + ')')

plt1.plot(np.arange(1, max_degree + 1, 1), J_train, '-', label='train')
plt1.plot(np.arange(1, max_degree + 1, 1), J_test, '-', label='test')
plt1.set_title('Loss vs polynomial degree')
plt1.set_xlabel('polynomial degree')
plt1.set_ylabel('loss')
plt1.grid(axis='both', alpha=.25)
plt1.legend()

plt2.set_title('Predicted monthly sales')
plt2.set_xlabel('Month')
plt2.set_ylabel('Sales ($)')
plt2.grid(axis='both', alpha=.25)
plt2.legend()
plt.show()
```



Take some time to undserstand the code. You should see that training loss falls as the degree of the polynomial increases. However, depending on your particular train/test split of the data, you may observe at $d=4$ or $d=5$ that test loss starts to flatten out or even increase. This is the phenomenon of overfitting!

If you don't see any evidence of overfitting, you might regenerate the test/train splits (comment out the seed setting in the partition function and re-run the rest of the cells, but don't forget to put the seed back before turning in your solution!).

You may also increase max_degree to a point. However, without normalization of the data, the matrix $\texttt{X}^\top\texttt{X}$ we invert in the solution to the normal equations will become numerically close to singularity, and you will observe unstable solutions. The result is usually a parameter vector θ that is suboptimal that gives poor results on both the training set and test set.

If you want to evaluate the numerial stability of the correlation matrix $\texttt{X}^\top\texttt{X}$, try this code:

In [201]:

```
corr = Xi_train.T.dot(Xi_train)
print('Correlation matrix:', corr)
cond = np.linalg.cond(corr)
print('Condition number: %0.5g' % cond)
```

```
Correlation matrix: [[1.44000000e+02 9.34000000e+02 7.73800000e+03 7.24
420000e+04
  7.25962000e+05 7.58679400e+06]
 [9.34000000e+02 7.73800000e+03 7.24420000e+04 7.25962000e+05
  7.58679400e+06 8.15402980e+07]
 [7.73800000e+03 7.24420000e+04 7.25962000e+05 7.58679400e+06
  8.15402980e+07 8.94004282e+08]
 [7.24420000e+04 7.25962000e+05 7.58679400e+06 8.15402980e+07
  8.94004282e+08 9.94854740e+09]
 [7.25962000e+05 7.58679400e+06 8.15402980e+07 8.94004282e+08
  9.94854740e+09 1.11986452e+11]
 [7.58679400e+06 8.15402980e+07 8.94004282e+08 9.94854740e+09
  1.11986452e+11 1.27211760e+12]]
Condition number: 6.5793e+12
```

Read more about the condition number on [Wikipedia](https://en.wikipedia.org/wiki/Condition_number). Roughly speaking, if our condition number is 10^k, we may lose up to k digits of accuracy in the inverse of the matrix. If $k=12$ as above, then we have an extremely poorly conditioned problem, because the IEEE 64 bit floating point representation of reals we're using in Python only has around 16 digits of accuracy (see [Wikipedia's page on IEEE floating point numbers](https://en.wikipedia.org/wiki/IEEE_754)).

One way to improve the numerical conditioning of the problem is normalization. If the values of the variables we are correlating in this matrix have relatively small positive and negative values, the condition number of the correlation matrix will be much smaller and you'll get better results.

In-lab exercises

During the lab session, you should perform the following exercises:

1. Add the `year` variable from the monthly sales dataset to your simple linear regression model and quantify whether including it improves test set performance. Show the observations and predictions in a 3D surface plot.
2. Develop polynomial regression models of degree 2 and 3 based on the two input variables. Show results as 3D surface plots and discuss whether you observe overfitting or not.

Exercise 2.1 (2 points)

Import **MonthlySales_data.csv** file into `data_csv` and extract **headers** at the top of `data_csv` into `headers_csv`.

In [202]:

```
# YOUR CODE HERE
import matplotlib.pyplot as plt
import numpy as np
import random

# Import CSV
data = np.genfromtxt('MonthlySales_data.csv',delimiter = ',', dtype=str)

# Extract headers
headers_csv = data[0,:]
headers = data[0,:]
print("Headers:", headers_csv)

# Extract raw data
data_csv = np.array(data[1:,:], dtype=float);
mean = np.mean(data_csv,axis=0)
std = np.std(data_csv,axis=0)
data_norm = (data_csv-mean)/std

# Extract y column from raw data
y_index = np.where(headers == 'sale amount')[0][0];
y_data = data_csv[:,y_index];

# Extract x column (just the month) from raw data
month_index = np.where(headers == 'month')[0][0]
# print(year_index, month_index)
X_data = data_csv[:,[month_index]];
m = X_data.shape[0]
n = X_data.shape[1]
X_data = X_data.reshape(m, n)

print('Extracted %d monthly sales records' % m)
print(X_data.shape)
print(y_data.shape)
```

```
Headers: ['year' 'month' 'sale amount']
Extracted 240 monthly sales records
(240, 1)
(240,)
```

In [203]:

```
print(headers_csv)
print(data_csv[:5])

# Test function: Do not remove
assert type(data_csv[0,0]) == np.float64, "You must remove the header"
assert headers_csv.shape[0] == 3, "Headers must have 3 values"
assert type(headers_csv[0]) == np.str_, "Headers must be string"
assert np.round(data_csv[30, 2], 3) == np.round(2.222027e+04, 3), "Data is incorrec
t"
print("success!")
# End Test function
```

```
['year' 'month' 'sale amount']
[[1.995000e+03 1.000000e+00 1.238611e+04]
 [1.995000e+03 2.000000e+00 1.532923e+04]
 [1.995000e+03 3.000000e+00 5.800217e+04]
 [1.995000e+03 4.000000e+00 5.130520e+04]
 [1.995000e+03 5.000000e+00 1.645247e+04]]
success!
```

Expected output:\ ['year' 'month' 'sale amount']\ [[1.995000e+03 1.000000e+00 1.238611e+04]\ [1.995000e+03 2.000000e+00 1.532923e+04]\ [1.995000e+03 3.000000e+00 5.800217e+04]\ [1.995000e+03 4.000000e+00 5.130520e+04]\ [1.995000e+03 5.000000e+00 1.645247e+04]]

Exercise 2.2 (2 points)

- Extract **sale amount** column into `y_csv`
- Extract **year** and **month** columns into `X_csv` by use **year** at column index 0 and **month** at column index 1

In [204]:

```
# Extract y column from raw data
# Extract x column (year and month) from raw data
# YOUR CODE HERE
y_csv = data_csv[:,2:]
X_csv = data_csv[:,0:2]
#raise NotImplementedError()
```

In [205]:

```
m = X_csv.shape[0]
n = X_csv.shape[1]
X_csv = X_csv.reshape(m, n)
print('Extracted %d sales records' % m)
print('number of x set:', n)

# Test function: Do not remove
assert m == 240, "Sales records incorrect"
assert n == 2, "Need to extract 2 columns of X set"
assert np.max(X_csv[:,0]) == 2014 and np.min(X_csv[:,0]) == 1995, "Year is filled w
rong column"
assert np.max(X_csv[:,1]) == 12 and np.min(X_csv[:,1]) == 1, "Month is filled wrong
column "
print("success")
# End Test function
```

```
Extracted 240 sales records
number of x set: 2
success
```

Expected output:\ Extracted 240 sales records\ number of x set: 2

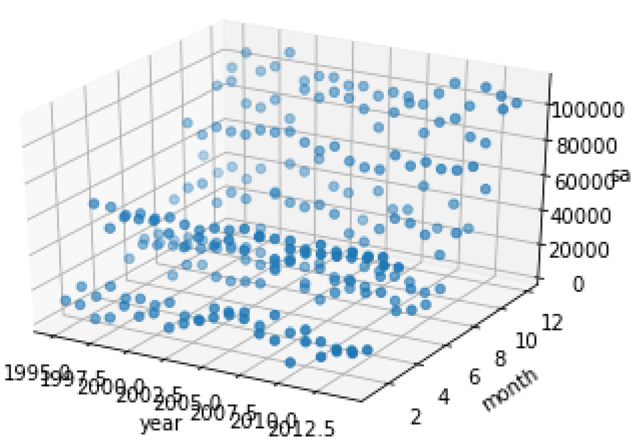
Exercise 2.3 (2 points)

Plot a 3D graph using the `mpl_toolkits.mplot3d` library.

Hint here!

In [206]:

```
# Plot the data
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
# 1. Set plot graph as 3D
ax = fig.add_subplot(projection='3d')

# 2. Extract data
# extract year at x-axis
# extract month at y-axis
# extract sale amount at z-axis
#x_year = None
#y_month = None
#z_sale = None
# 3. plot by using scatter
# 4. set x, y, z label
# YOUR CODE HERE
x_year = X_csv[:,0:1]
y_month = X_csv[:,1:2]
z_sale = y_csv
ax.scatter(x_year, y_month, z_sale)

ax.set_xlabel('year')
ax.set_ylabel('month')
ax.set_zlabel('sale')

plt.show()
```



In [207]:

```
# Test function: Do not remove
assert ax.get_xbound()[1] >= 2014 and ax.get_xbound()[0] <= 1995, "Year is filled w
rong column"
assert ax.get_ybound()[1] >= 12 and ax.get_ybound()[0] <= 1, "Month is filled wrong
column"
assert ax.get_zbound()[1] >= 100000 and ax.get_zbound()[0] <= 0, "Year is filled wr
ong column"
assert 'year' in ax.get_xlabel().lower(), "x-axis label is incorrect"
assert 'month' in ax.get_ylabel().lower(), "y-axis label is incorrect"
assert 'sale' in ax.get_zlabel().lower(), "y-axis label is incorrect"
print("success")
# End Test function
```

```
success
```

Expected output:

Exercise 2.4 (2 points)

Extract 60% of the data to the training set and the remaining 40% to the test set with shuffling.

You can use the `partitions` function we already made or create a new function. Make sure that you use `random.seed(1412)` to make sure that the result is the same as the expect result. Place the resulting data in variables `idx, X_train, y_train, X_test, y_test`.

In [208]:

```
# YOUR CODE HERE
percent_train = .6
def partition(X, y, percent_train):
    # Create a list of indices into X and y
    idx = np.arange(0,y.shape[0])
    random.seed(1412)
    # just make sure the shuffle always the same please do not remove
    # On your own, do the following:
    # 1. shuffle the idx list
    # 2. Create lists of indices train_idx and test_idx for the train and test sets
    # 3. Set variables X_train, y_train, X_test, and y_test using those index lists

    # YOUR CODE HERE
    random.shuffle(idx)
    size_train=int(y.shape[0]*percent_train)
    train_idx=idx[0:size_train]
    test_idx=idx[size_train:y.shape[0]+1]

    #X_train = data[train_idx,0:y_index]
    X_train= X[train_idx,:]
    #X_test = data[test_idx,2:y_index]
    X_test = X[test_idx,:]

    #y_train = data[train_idx,y_index]
    y_train=y[train_idx,]
    #y_test = data[test_idx,y_index]
    y_test=y[test_idx,]
    return idx, X_train, y_train, X_test, y_test
idx, X_train, y_train, X_test, y_test = partition(X_csv, y_csv, percent_train)
```

In [209]:

```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
print(idx[5:9])

# Test function: Do not remove
assert not np.array_equal(np.round(X_csv[0:144, :], 3), np.round(X_train,3)), "X_tr
ain must be shuffled!"
assert not np.array_equal(np.round(X_csv[144:, :], 3), np.round(X_test,3)), "X_test
must be shuffled!"
assert not np.array_equal(np.round(y_csv[0:144], 3), np.round(y_train,3)), "y_train
must be shuffled!"
assert not np.array_equal(np.round(y_csv[144:], 3), np.round(y_test,3)), "y_test mu
st be shuffled!"
assert np.array_equal(idx[5:9], [26, 75, 51, 162])
print("success!")
# End Test function
```

```
(144, 2)
(144, 1)
(96, 2)
(96, 1)
[ 26  75  51 162]
success!
```

Expected output:\ (144, 2)\ (144,)\ (96, 2)\ (96,)\ [26 75 51 162]

Exercise 2.5 (2 points)

1. Create `Xi_train, Xi_Test`. X sets must be polynomial of $n=1$.
2. Calculate `theta`
3. Calculate `y_pred_test`
4. Calculate cost function J from train and test set

In [212]:

```
def x_polynomial(x, d):
    # YOUR CODE HERE
    X = np.array([x]).T
    X = np.insert(x, 0, 1, axis=1)

    while X.shape[1] < d+1:
        #print(np.shape(X))
        newcol = X[:,-1:]*X
        X=np.insert(X,X.shape[1],newcol.T,axis=1)
    return X

def regression(X, y):
    cov = np.dot(X.T, X)
    #print(cov.shape)
    cov_inv = np.linalg.inv(cov)
    theta = np.dot(cov_inv, np.dot(X.T, y))
    theta=theta.reshape(3,)
    #theta=np.around(theta,5).T
    return theta

def h(X, theta):
    return X.dot(theta)

def cost(theta, X, y):
    # YOUR CODE HERE
    thetaX=h(X, theta)#.reshape(1,3)
    sum_function= (y.T-thetaX)**2
    fo=(1/(2*y.shape[0]))
    #J = (1/(2*y.shape[0]))*np.sum(sum_function**2)
    J = fo*np.sum(sum_function)
    #print(J)
    return J

# YOUR CODE HERE
Xi_train = x_polynomial(X_train, 1)
Xi_test = x_polynomial(X_test, 1)
theta = regression(Xi_train, y_train)

# calculate cost in train
J_train = cost(theta, Xi_train, y_train)

y_pred_test = h(Xi_test, theta)
J_test = cost(theta, Xi_test, y_test)
J_train = cost(theta, Xi_train, y_train)
```

In [213]:

```
print("Xi_train[:3]:", np.round(Xi_train[:3], 2))
print("Xi_test[:3]:", np.round(Xi_test[:3], 2))
print("theta:", theta)
print("y_pred_test[:5]:", np.round(y_pred_test[:5].T, 2))
print("J_train:", J_train)
print("J_test:", J_test)

# Test function: Do not remove
assert np.array_equal(np.round(theta, 3), np.round([5.74503812e+05, -2.83158807e+02
, 6.37579347e+03],3)), "Regression theta is incorrect"
assert np.round(J_train, 0) == np.round(172968387.44854635, 0), "Train cost is inco
rrect"
assert np.round(J_test, 0) == np.round(204275431.7643744, 0), "Test cost is incorre
ct"
print("success")
# End Test function
```

```
Xi_train[:3]: [[1.000e+00 2.003e+03 1.100e+01]
 [1.000e+00 2.004e+03 3.000e+00]
 [1.000e+00 2.002e+03 6.000e+00]]
Xi_test[:3]: [[1.000e+00 2.008e+03 1.000e+01]
 [1.000e+00 1.997e+03 5.000e+00]
 [1.000e+00 2.006e+03 1.100e+01]]
theta: [ 5.74503812e+05 -2.83158807e+02  6.37579347e+03]
y_pred_test[:5]: [69678.86 40914.64 76620.97 79169.4  48852.53]
J_train: 172968387.44854638
J_test: 204275431.76439014
success
```

Expected output:\ Xi_train[:3]: [[1.000e+00 2.003e+03 1.100e+01]\ [1.000e+00 2.004e+03 3.000e+00]\ [1.000e+00 2.002e+03 6.000e+00]]\ Xi_test[:3]: [[1.000e+00 2.008e+03 1.000e+01]\ [1.000e+00 1.997e+03 5.000e+00]\ [1.000e+00 2.006e+03 1.100e+01]]\ theta: [5.74503812e+05 -2.83158807e+02 6.37579347e+03]\ y_pred_test[:5]: [69678.86 40914.64 76620.97 79169.4 48852.53]\ J_train: 172968387.44854635\ J_test: 204275431.7643744

Exercise 2.6 (2 points)

Create a mesh of grid points in order to obtain a surface plot later.

Hint here!

In [214]:

```
x_year = X_csv[:,0:1]
y_month = X_csv[:,1:2]
z_sale = y_csv
# 1. Create mesh grid x_mesh, y_mesh
#    Hint: this step do in input X dataset only (year, and month series)
# 1.1 use numpy.linspace() to generate x_series and y_series
#     - do x_series in between min(year) - 1 to max(year) + 1
minX_1=np.amin(x_year)-1
maxX_1=np.amax(x_year)+1
x_series=np.linspace(minX_1,maxX_1,num=100)

#    - do y_series in between min(month) - 1 to max(month) + 1

minY_1=np.amin(y_month)-1
maxY_1=np.amax(y_month)+1
y_series=np.linspace(minY_1,maxY_1,num=100)

#     - num_linspace = 100
# 1.2 use numpy.meshgrid() to generate x_mesh, and y_mesh
# 1.3 merge x_mesh and y_mesh to be xy_mesh
num_linspace = 100

x_mesh, y_mesh = np.meshgrid(x_series, y_series)

#x_mesh, y_mesh = np.meshgrid(x_series, y_series)

#xy_mesh = np.array(np.meshgrid(x_mesh,y_mesh)).T
xy_mesh = np.dstack((x_mesh, y_mesh))
#print("xy_mesh.shape", xy_mesh.shape)
#print("xy_mesh.shape", xy_mesh.shape)

# 2. predict output from xy_mesh to be z_series
#    Hint: use mesh_predictions function instead of get_prediction
def mesh_predictions(x, theta):
    x = np.insert(x, 0, 1, axis=x.ndim-1)
    theta = theta.reshape(-1,1)
    y = x@theta
    print(x.shape)
    print(theta.shape)
    return y

z_series = mesh_predictions(xy_mesh, theta)
z_series=z_series.reshape(z_series.shape[0],z_series.shape[1])
#print(z_series)
# YOUR CODE HERE
#minZ_1=np.amin(xy_mesh)-1
#maxZ_1=np.amax(xy_mesh)+1
#z_series=np.linspace(minZ_1,maxZ_1)
```

```
(100, 100, 3)
(3, 1)
```

In [215]:

```
print("xy_mesh.shape", xy_mesh.shape)
print("z_series.shape", z_series.shape)
#print("xy_mesh", xy_mesh)
#print("z_series", z_series)

# Test function: Do not remove
assert xy_mesh.shape == (num_linspace, num_linspace, 2), "mesh shape is incorrect"
assert z_series.shape == (num_linspace, num_linspace), "z_series is incorrect"
print("success")
# End Test function
```

```
xy_mesh.shape (100, 100, 2)
z_series.shape (100, 100)
success
```

Expected output:\ xy_mesh.shape (100, 100, 2)\ z_series.shape (100, 100)

Exercise 2.6 (2 points)

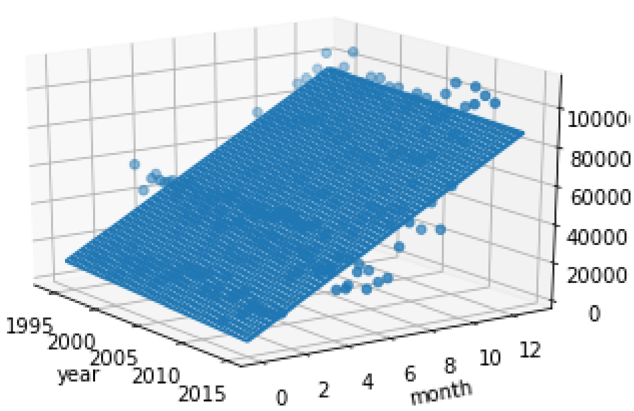
Make a surface plot for theta with the dataset points from `xy_mesh` and `z_series` variables created above.

Hint here!

In [216]:

```
fig = plt.figure()
# 1. Set plot graph as 3D
ax = fig.add_subplot(projection='3d')

# 2. Extract data
# extract year at x-axis
# extract month at y-axis
# extract sale amount at z-axis

x_year = X_csv[:,0:1]
y_month = X_csv[:,1:2]
z_sale = y_csv
ax.scatter(x_year, y_month, z_sale)
ax.set_xlabel('year')
ax.set_ylabel('month')
ax.set_zlabel('sale')
print(z_series.shape)
ax.view_init(elev=20., azim=-35)
ax.plot_wireframe(x_mesh,y_mesh, z_series)

# 3. plot by using scatter
# 4. set x, y, z label
#    Hint: In these 3, 4 steps, you can copy Exercise 2.3
# 5. Plot surface from x_mesh, y_mesh, and z_series

# YOUR CODE HERE


plt.show()
```

```
(100, 100)
```



In [217]:

```
# Test function: Do not remove
assert ax.get_xbound()[1] >= 2014 and ax.get_xbound()[0] <= 1995, "Year is filled w
rong column"
assert ax.get_ybound()[1] >= 12 and ax.get_ybound()[0] <= 1, "Month is filled wrong
column"
assert ax.get_zbound()[1] >= 100000 and ax.get_zbound()[0] <= 0, "Year is filled wr
ong column"
assert 'year' in ax.get_xlabel().lower(), "x-axis label is incorrect"
assert 'month' in ax.get_ylabel().lower(), "y-axis label is incorrect"
assert 'sale' in ax.get_zlabel().lower(), "y-axis label is incorrect"
print("success")
# End Test function
```

```
success
```

Expect result:

Exercise 2.7 (20 points)

Develop polynomial regression models of degree 2 and 3 based on the two input variables. Show results as 3D surface plots and discuss whether you observe overfitting or not.

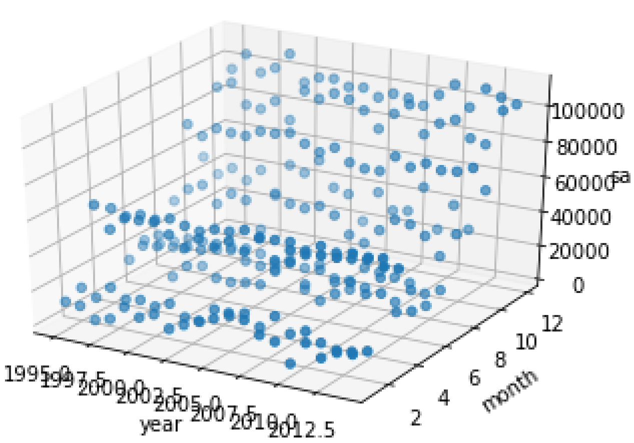
Discuss : When I try to add more degree at 2 and 3. I observe about in highter degree will fiting bestter. For overfiting i do not observe sing of that because i see some group of data out of prediced

In [244]:

```
import matplotlib.pyplot as plt
import numpy as np
import random

# Import CSV
data = np.genfromtxt('MonthlySales_data.csv',delimiter = ',', dtype=str)

# Extract headers
headers_csv = data[0,:]
headers = data[0,:]
print("Headers:", headers_csv)

# Extract raw data
data_csv = np.array(data[1:,:], dtype=float);
mean = np.mean(data_csv,axis=0)
std = np.std(data_csv,axis=0)
data_norm = (data_csv-mean)/std

# Extract y column from raw data
y_index = np.where(headers == 'sale amount')[0][0];
y_data = data_csv[:,y_index];

# Extract x column (just the month) from raw data
month_index = np.where(headers == 'month')[0][0]
# print(year_index, month_index)
X_data = data_csv[:,[month_index]];
m = X_data.shape[0]
n = X_data.shape[1]
X_data = X_data.reshape(m, n)

print('Extracted %d monthly sales records' % m)
print(X_data.shape)
print(y_data.shape)
# Extract y column from raw data
# Extract x column (year and month) from raw data
# YOUR CODE HERE
y_csv = data_csv[:,2:]
X_csv = data_csv[:,0:2]
#raise NotImplementedError()
m = X_csv.shape[0]
n = X_csv.shape[1]
X_csv = X_csv.reshape(m, n)
print('Extracted %d sales records' % m)
print('number of x set:', n)

# Test function: Do not remove
assert m == 240, "Sales records incorrect"
assert n == 2, "Need to extract 2 columns of X set"
assert np.max(X_csv[:,0]) == 2014 and np.min(X_csv[:,0]) == 1995, "Year is filled w
rong column"
assert np.max(X_csv[:,1]) == 12 and np.min(X_csv[:,1]) == 1, "Month is filled wrong
column "
print("success")
# End Test function
# Plot the data
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
# 1. Set plot graph as 3D
ax = fig.add_subplot(projection='3d')

# 2. Extract data
# extract year at x-axis
# extract month at y-axis
# extract sale amount at z-axis
#x_year = None
#y_month = None
#z_sale = None
# 3. plot by using scatter
# 4. set x, y, z label
# YOUR CODE HERE
x_year = X_csv[:,0:1]
```

```python
y_month = X_csv[:,1:2]
z_sale = y_csv
ax.scatter(x_year, y_month, z_sale)

ax.set_xlabel('year')
ax.set_ylabel('month')
ax.set_zlabel('sale')

plt.show()
# YOUR CODE HERE
percent_train = .6
def partition(X, y, percent_train):
    # Create a list of indices into X and y
    idx = np.arange(0,y.shape[0])
    random.seed(1412)
    # just make sure the shuffle always the same please do not remove
    # On your own, do the following:
    # 1. shuffle the idx list
    # 2. Create lists of indices train_idx and test_idx for the train and test sets
    # 3. Set variables X_train, y_train, X_test, and y_test using those index lists

    # YOUR CODE HERE
    random.shuffle(idx)
    size_train=int(y.shape[0]*percent_train)
    train_idx=idx[0:size_train]
    test_idx=idx[size_train:y.shape[0]+1]

    #X_train = data[train_idx,0:y_index]
    X_train= X[train_idx,:]
    #X_test = data[test_idx,2:y_index]
    X_test = X[test_idx,:]

    #y_train = data[train_idx,y_index]
    y_train=y[train_idx,]
    #y_test = data[test_idx,y_index]
    y_test=y[test_idx,]
    return idx, X_train, y_train, X_test, y_test
def x_polynomial(x, d):
    # YOUR CODE HERE
    X = np.insert(x, 0, x[:,0]**0,axis=1)
    for j in range(2,d+1):
        X = np.insert(X, [X.shape[1]], x**j,axis=1)
    return X
```

```
Headers: ['year' 'month' 'sale amount']
Extracted 240 monthly sales records
(240, 1)
(240,)
Extracted 240 sales records
number of x set: 2
success
```



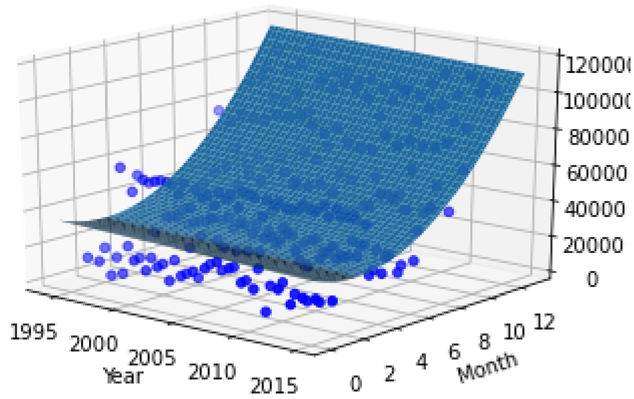
In [245]:

```
idx, X_train, y_train, X_test, y_test = partition(X_csv, y_csv, percent_train)
```

In [246]:

```
def regression(X, y):
    cov = np.dot(X.T, X)
    print(cov.shape)
    cov_inv = np.linalg.inv(cov)
    theta = np.dot(cov_inv, np.dot(X.T, y))
    theta=theta.reshape(theta.shape[0],)
    #theta=np.around(theta,5).T
    return theta
def h(X, theta):
    return X.dot(theta)
def cost(theta, X, y):
    # YOUR CODE HERE
    sum_function= y-(np.dot(theta,X.T))
    J = (1/(2*y.shape[0]))*np.sum(sum_function**2)
    return J
def mesh_predictions(x, theta):
    x = np.insert(x, 0, 1, axis=x.ndim-1)
    theta = theta.reshape(-1,1)
    y = x@theta
    return y


Xi_train= x_polynomial(X_train, 2)
Xi_test= x_polynomial(X_test, 2)

theta = regression(Xi_train, y_train)
y_pred_test = h(Xi_test, theta)
J_train= cost(theta, Xi_train, y_train)
J_test = cost(theta, Xi_test, y_test)

num_linspace = 100
x_series = np.linspace(min(x_year) - 1, max(x_year) + 1, num_linspace)
y_series= np.linspace(min(y_month) - 1, max(y_month) + 1, num_linspace)

x_mesh, y_mesh = np.meshgrid(x_series, y_series)
xy_mesh = np.stack((x_mesh, y_mesh),axis=2)
xy_mesh_d2 = xy_mesh**2
last_xy_mesh = np.concatenate((xy_mesh, xy_mesh_d2), axis=2)


z_series = mesh_predictions(last_xy_mesh, theta).reshape(num_linspace,-1)
fig = plt.figure()

ax = fig.add_subplot(projection='3d')

x_year = x_year
y_month = y_month
z_sale = y_csv


ax.scatter(x_year, y_month, z_sale,color='blue')
ax.set_xlabel('Year')
ax.set_ylabel('Month')
ax.set_zlabel('Sale Amount')
ax.plot_surface(x_mesh, y_mesh,z_series)
ax.view_init(elev=20., azim=-50)
plt.show()
```

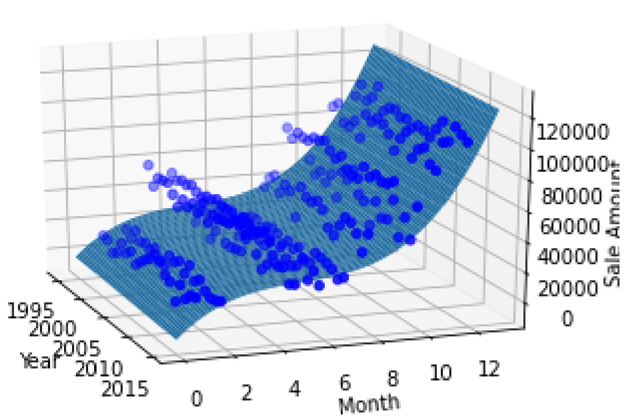
(5, 5)


120000
100000
80000
60000
40000
20000
0
1995
2000
2005
2010
2015
Year
0
2
4
6
8
10
12
Month

In [247]:

```
Xi_train= x_polynomial(X_train, 3)
Xi_test = x_polynomial(X_test, 3)
theta = regression(Xi_train, y_train)
y_pred_test = h(Xi_test, theta)
J_train= cost(theta, Xi_train, y_train)
J_test =  cost(theta, Xi_test, y_test)
num_linspace = 100
x_series  = np.linspace(min(x_year) - 1, max(x_year) + 1, num_linspace)
y_series = np.linspace(min(y_month) - 1, max(y_month) + 1, num_linspace)

x_mesh, y_mesh = np.meshgrid(x_series, y_series)
xy_mesh = np.stack((x_mesh, y_mesh),axis=2)
xy_mesh_d2 = xy_mesh**2
xy_mesh_d3 = xy_mesh**3
last_xy_mesh = np.concatenate((xy_mesh, xy_mesh_d2), axis=2)
last_xy_mesh = np.concatenate((last_xy_mesh, xy_mesh_d3), axis=2)


def mesh_predictions(x, theta):
    x = np.insert(x, 0, 1, axis=x.ndim-1)
    theta = theta.reshape(-1,1)
    y = x@theta
    return y
z_series = mesh_predictions(last_xy_mesh, theta).reshape(num_linspace,-1)
fig = plt.figure()

ax = fig.add_subplot(projection='3d')


x_year = x_year
y_month = y_month
z_sale = y_csv


ax.scatter(x_year, y_month, z_sale,color='blue')
ax.set_xlabel('Year')
ax.set_ylabel('Month')
ax.set_zlabel('Sale Amount')
ax.plot_surface(x_mesh, y_mesh,z_series)
ax.view_init(elev=20., azim=-20)
plt.show()
```

```
(7, 7)
```



Discuss : When I try to add more degree at 2 and 3. I observe about in highter degree will fiting bestter. For overfiting i do not observe sing of that because i see some group of data out of prediced

Take-home exercise (50 points)

Using the dataset you played with for the take-home exercise in Lab 01, perform the same analysis. You won't be able to visualize the model well, as you will have more than two inputs, but try to give some idea of the performance of the model visually. Also, depending on the number of variables in your dataset, you may not be able to increase the polynomial degree beyond 2. Discuss whether the polynomial model is better than the linear model and whether you observe overfitting.

Insert your code, explanation, and results here.

To turn in

Before the next lab, turn in a brief report in the form of a Jupyter notebook documenting your work in the lab and the take-home exercise, along with your observations and discussion.

Datasets that,I am interesting is Average age people in a country lived. (https://www.kaggle.com/brendan45774/countries-life-expectancy (https://www.kaggle.com/brendan45774/countries-life-expectancy)) The data collect for 15 country from 1800 to 2016. That have 217 rows × 2 columns by columns of Year that get the data and Life expectancy. I calculate with Simple linear regression, polynomial regression models in degree 2, 3 , 4 , 5 and 6. The result fo Simple linear regression is theta1: [0.04203186 0.9261063] J_train: 0.06261 J_test: 0.06076.

For the result of polynomial in degree 2 is theta: [-0.33543971 0.93653617 0.34101305] J_train: 0.01402 J_test: 0.01085, accuracy : 0.8590.

For the result of polynomial in degree 3 is theta: [-0.34480829 1.14561884 0.34180978 -0.11203524] J_train: 0.00997 J_test: 0.00713, accuracy : 0.9748.

For the result of polynomaial in degree 4 is theta: [-0.4768604 1.14393485 0.748639 -0.11499635 -0.15422021] J_train: 0.0042972 J_test: 0.0030887, accuracy : 0.9928.

For the result of polynomaial in degree 5 is theta: [-0.4768604 1.14393485 0.748639 -0.11499635 -0.15422021] J_train: 0.004295 J_test: 0.00307, accuracy : 0.99286.

For the result of polynomaial in degree 6 is theta: [-0.52070011 1.15064805 1.02118005 -0.12623241 -0.41428202 0.00362973 0.06198735] J_train: 0.00373 J_test: 0.00335, accuracy : 0.992.

In conclusion, When I calculation in hight degree will get lower J. How about graph line the perdiced by theta from hight degree model will fiting bester than lower degree about accuracy hight degree will give hight accuracy. However affter i try give more degree will face with overfitting at polynomaial in degree 5 bacause accuracy change from polynomaial in degree 4 to polynomaial in degree 5 just 0.00006. it is to small. Than we should to stop to calculation at polynomaial in degree 5.

In [222]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np
from time import time
import math,random
from numpy.random import default_rng
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
```

In [223]:

```
data_raw = pd.read_csv('Life expectancy.csv')
df=pd.DataFrame(data_raw)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3253 entries, 0 to 3252
Data columns (total 3 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Entity           3253 non-null   object 
 1   Year             3253 non-null   int64  
 2   Life expectancy  3253 non-null   float64
dtypes: float64(1), int64(1), object(1)
memory usage: 76.4+ KB
```

In [224]:

```
df.isnull().sum()
```

Out[224]:

```
Entity             0
Year               0
Life expectancy    0
dtype: int64
```

In [225]:

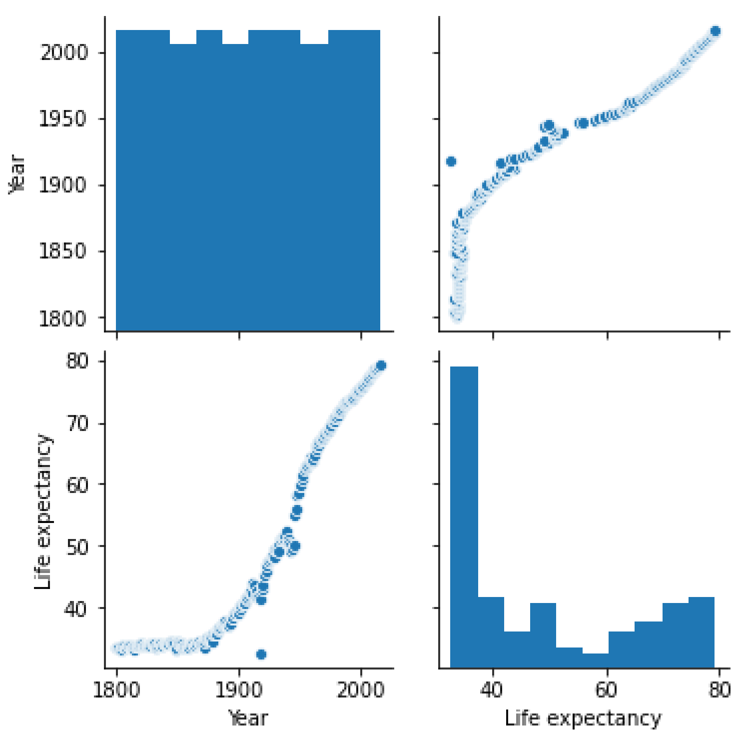
```
df=df.groupby('Year').mean().reset_index()  #I use mean of Life expectancy that mea
n of Life expectancy of people of the world
```

In [226]:

```
sns.pairplot(df)
```

Out[226]:

```
<seaborn.axisgrid.PairGrid at 0x7fd95910e610>
```

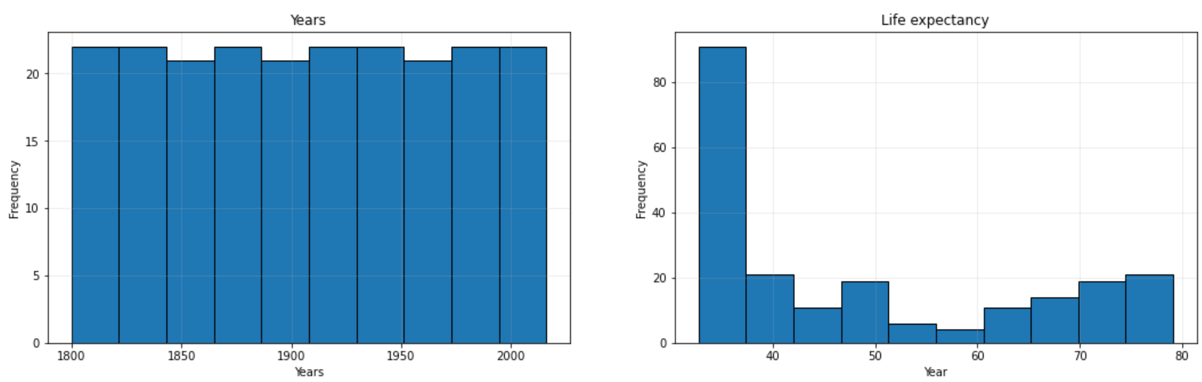


In [227]:

```
data=df.to_numpy()
```

In [228]:

```
# Visualise the distribution of independent and dependent variables

fig, ax = plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(20)
fig.subplots_adjust(left=.2, bottom=None, right=None, top=None, wspace=.2, hspace=.
2)
plt1 = plt.subplot(1,2,1)
plt2 = plt.subplot(1,2,2)


# Years
plt1.hist(data[:,0], label='Years', edgecolor='black')
plt1.set_title('Years')
plt1.set_xlabel('Years')
plt1.set_ylabel('Frequency')
plt1.grid(axis='both', alpha=.25)

# Variable 2: Life expectancy
plt2.hist(data[:,1], label='Life expectancy', edgecolor='black')
plt2.set_title('Life expectancy')
plt2.set_xlabel('Year')
plt2.set_ylabel('Frequency')
plt2.grid(axis='both', alpha=.25)
```



In [229]:

```
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
data_norm = (data - means) / stds
print(data_norm.shape)
```

```
(217, 2)
```

In [230]:

```
#y_index = 2
y = data_norm[:,1]

# Extract X from normalized dataset
y_index=1
X = data_norm[:,0]

def partition(X, y, percent_train):
    # Create a list of indices into X and y
    idx = np.arange(0,y.shape[0])
    random.seed(1412)
    # just make sure the shuffle always the same please do not remove
    # On your own, do the following:
    # 1. shuffle the idx list
    # 2. Create lists of indices train_idx and test_idx for the train and test sets
    # 3. Set variables X_train, y_train, X_test, and y_test using those index lists

    # YOUR CODE HERE
    random.shuffle(idx)
    size_train=int(y.shape[0]*percent_train)
    train_idx=idx[0:size_train]
    test_idx=idx[size_train:y.shape[0]+1]

    X_train = X[train_idx]
    X_train=X_train.reshape(X_train.shape[0],1)
    X_test = X[test_idx]
    X_test=X_test.reshape(X_test.shape[0],1)

    y_train = y[train_idx]
    y_test = y[test_idx]

    return idx, X_train, y_train, X_test, y_test
def regression(X, y):
    cov = np.dot(X.T, X)
    print(cov.shape)
    cov_inv = np.linalg.inv(cov)
    theta = np.dot(cov_inv, np.dot(X.T, y))
    theta=theta.reshape(theta.shape[0],)
    #theta=np.around(theta,5).T
    return theta
def h(X, theta):
    return X.dot(theta)
```

In [231]:

```
idx, X_train, y_train, X_test, y_test = partition(X, y, 0.6)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(130, 1)
(130,)
(87, 1)
(87,)
```

In [232]:

```
def x_polynomial(x, d):
    # YOUR CODE HERE
    X = np.array([x]).T
    X = np.insert(x, 0, 1, axis=1)
    while X.shape[1] < d+1:
        newcol = X[:,-1:]*x
        X=np.insert(X,X.shape[1],newcol.T,axis=1)
    return X
```

In [233]:

```
Xi_train = x_polynomial(X_train, 2)
Xi_test = x_polynomial(X_test, 2)
```

In [234]:

```
def cost(theta, X, y):
    # YOUR CODE HERE
    sum_function= y-(np.dot(theta,X.T))
    J = (1/(2*y.shape[0]))*np.sum(sum_function**2)
    return J
def get_predictions(x, theta):
    # Change the shape of x to support the function
    x = np.array([x]).T
    X_aug = np.insert(x, 0, 1, axis=1)
    while X_aug.shape[1] < theta.shape[0]:
        newcol = X_aug[:,-1:]*x
        X_aug=np.insert(X_aug,X_aug.shape[1],newcol.T,axis=1)

    y_hat = X_aug@theta
    return y_hat
def r_squared(y, y_pred):
    # YOUR CODE HERE
    up=np.sum((y-y_pred)**2)
    low = np.sum((y-np.mean(y))**2)
    r_sqr = 1-(up/low)
    #raise NotImplementedError()
    return r_sqr
```

In [235]:

```
# calculate theta
theta = regression(Xi_train, y_train)

# calculate cost in train
J_train = cost(theta, Xi_train, y_train)

y_pred_test = h(Xi_test, theta)
J_test = cost(theta, Xi_test, y_test)
print("theta2:", theta)
print("J_train:", J_train)
print("J_test:", J_test)
```

```
(3, 3)
theta2: [-0.33543971  0.93653617  0.34101305]
J_train: 0.01401586052747789
J_test: 0.010847748238723792
```

In [236]:

```
Xi_train1 = x_polynomial(X_train, 1)
Xi_test1 = x_polynomial(X_test, 1)
# calculate theta
theta1 = regression(Xi_train1, y_train)

# calculate cost in train
J_train1 = cost(theta1, Xi_train1, y_train)

y_pred_test1 = h(Xi_test1, theta1)
J_test1 = cost(theta1, Xi_test1, y_test)
print("theta1:", theta1)
print("J_train1:", J_train1)
print("J_test1:", J_test1)
```

```
(2, 2)
theta1: [0.04203186 0.9261063 ]
J_train1: 0.06260276774995922
J_test1: 0.060755532598164616
```

In [237]:

```
Xi_train3 = x_polynomial(X_train, 3)
Xi_test3 = x_polynomial(X_test, 3)
# calculate theta
theta3 = regression(Xi_train3, y_train)

# calculate cost in train
J_train3 = cost(theta3, Xi_train3, y_train)

y_pred_test3 = h(Xi_test3, theta3)
J_test3 = cost(theta3, Xi_test3, y_test)
print("theta4:", theta3)
print("J_train3:", J_train3)
print("J_test3:", J_test3)
```

```
(4, 4)
theta4: [-0.34480829  1.14561884  0.34180978 -0.11203524]
J_train3: 0.009969543138493082
J_test3: 0.007133023905595831
```

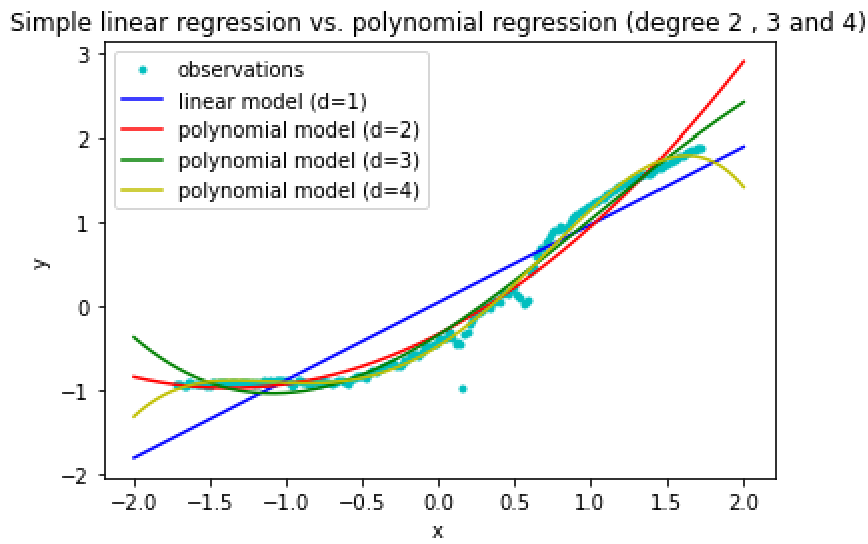
In [238]:

```
Xi_train4 = x_polynomial(X_train, 4)
Xi_test4 = x_polynomial(X_test, 4)
# calculate thet
theta4 = regression(Xi_train4, y_train)

# calculate cost in train
J_train4 = cost(theta4, Xi_train4, y_train)

y_pred_test4 = h(Xi_test4, theta4)
J_test4 = cost(theta4, Xi_test4, y_test)
print("theta4:", theta4)
print("J_train4:", J_train4)
print("J_test4:", J_test4)
```

```
(5, 5)
theta4: [-0.4768604   1.14393485  0.748639   -0.11499635 -0.15422021]
J_train4: 0.004297187710696183
J_test4: 0.0030887121562261227
```

In [239]:

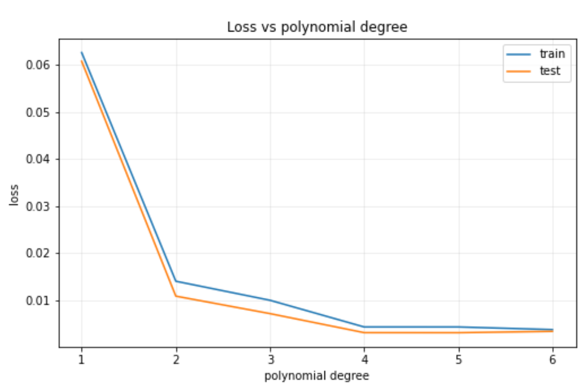
```
x_series = np.linspace(-2, 2, 1000)
y_series_slr = get_predictions(x_series, theta1)
y_series_pr = get_predictions(x_series, theta)
y_series_t = get_predictions(x_series, theta3)
y_series_f = get_predictions(x_series, theta4)
plt.plot(X, y, 'c.', label='observations')
plt.plot(x_series, y_series_slr, 'b-', label='linear model (d=1)')
plt.plot(x_series, y_series_pr, 'r-', label='polynomial model (d=2)')
plt.plot(x_series, y_series_t, 'g-', label='polynomial model (d=3)')
plt.plot(x_series, y_series_f, 'y-', label='polynomial model (d=4)')
plt.title('Simple linear regression vs. polynomial regression (degree 2 , 3 and 4)'
)
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

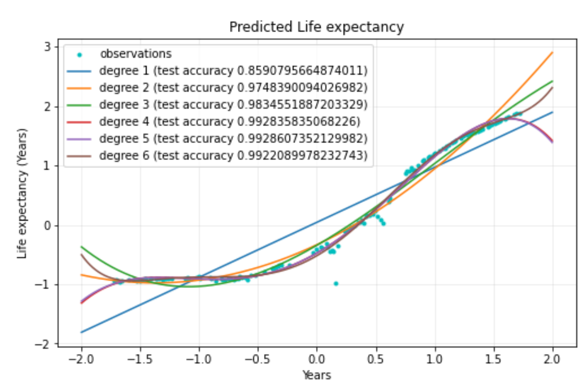


In [241]:

```
max_degree = 6

J_train = np.zeros(max_degree)
J_test = np.zeros(max_degree)

# Initalize plots for predictions and loss
fig, ax = plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(20)
fig.subplots_adjust(left=.2, bottom=None, right=None, top=None, wspace=.2, hspace=.
2)
plt1 = plt.subplot(1,2,1)
plt2 = plt.subplot(1,2,2)
plt2.plot(X_train, y_train, 'c.', label='observations')

for i in range(1, max_degree+1):
    # Fit model on training data and get cost for training and test data
    Xi_train = x_polynomial(X_train, i)
    Xi_test = x_polynomial(X_test, i);
    theta = regression(Xi_train, y_train)
    J_train[i-1] = cost(theta, Xi_train, y_train)
    y_pred_test = h(Xi_test, theta)
    J_test[i-1] = cost(theta, Xi_test, y_test)

    # Plot
    x_series = np.linspace(-2, 2, 1000)
    y_series = get_predictions(x_series, theta)
    plt2.plot(x_series, y_series, '-', label='degree ' + str(i) + ' (test accuracy
 ' + str(r_squared(y_test, y_pred_test)) + ')')
    print('test accuracy {0} : {1} '.format(i,r_squared(y_test, y_pred_test)))
    print('J_train {0} : {1} '.format(i,cost(theta, Xi_train, y_train)))
    print('J_test {0} : {1} '.format(i,cost(theta, Xi_test, y_test)))
    print('theta {0} : {1} '.format(i,theta))
plt1.plot(np.arange(1, max_degree + 1, 1), J_train, '-', label='train')
plt1.plot(np.arange(1, max_degree + 1, 1), J_test, '-', label='test')
plt1.set_title('Loss vs polynomial degree')
plt1.set_xlabel('polynomial degree')
plt1.set_ylabel('loss')
plt1.grid(axis='both', alpha=.25)
plt1.legend()

plt2.set_title('Predicted Life expectancy')
plt2.set_xlabel('Years')
plt2.set_ylabel('Life expectancy (Years)')
plt2.grid(axis='both', alpha=.25)
plt2.legend()
plt.show()
```

```
(2, 2)
test accuracy 1 : 0.8590795664874011
J_train 1 : 0.06260276774995922
J_test 1 : 0.060755532598164616
theta 1 : [0.04203186 0.9261063 ]
(3, 3)
test accuracy 2 : 0.9748390094026982
J_train 2 : 0.01401586052747789
J_test 2 : 0.010847748238723792
theta 2 : [-0.33543971  0.93653617  0.34101305]
(4, 4)
test accuracy 3 : 0.9834551887203329
J_train 3 : 0.009969543138493082
J_test 3 : 0.007133023905595831
theta 3 : [-0.34480829  1.14561884  0.34180978 -0.11203524]
(5, 5)
test accuracy 4 : 0.992835835068226
J_train 4 : 0.004297187710696183
J_test 4 : 0.0030887121562261227
theta 4 : [-0.4768604   1.14393485  0.748639   -0.11499635 -0.15422021]
(6, 6)
test accuracy 5 : 0.9928607352129982
J_train 5 : 0.004295140670753582
J_test 5 : 0.0030779768673848073
theta 5 : [-0.4764755   1.13666179  0.74802048 -0.10400629 -0.15406814
-0.00325239]
(7, 7)
test accuracy 6 : 0.9922089978232743
J_train 6 : 0.003734468406137646
J_test 6 : 0.003358962748848705
theta 6 : [-0.52070011  1.15064805  1.02118005 -0.12623241 -0.41428202
0.00362973
  0.06198735]
```





In [ ]: