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 [1]:

```
NAME = "Aung Zar Lin"
ID = "121956"
```

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 $\theta$, allowing us to use the normal equations to find the optimal $\theta$ 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 [2]:

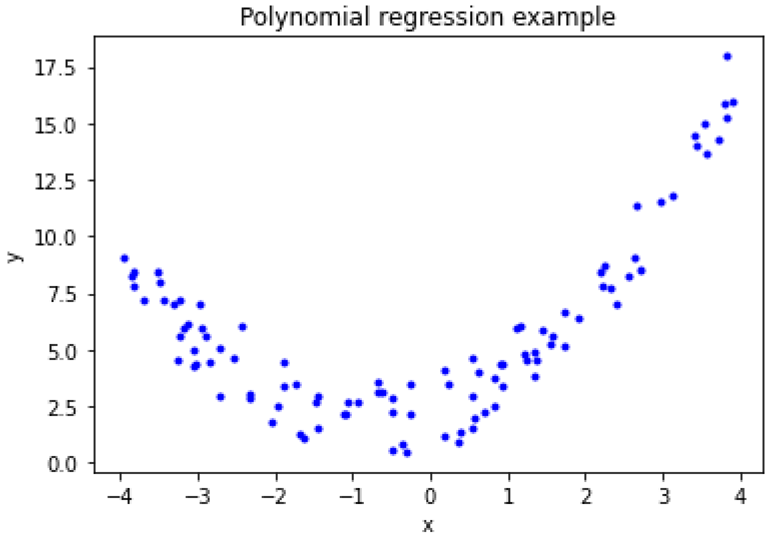
```
import matplotlib.pyplot as plt
import numpy as np
import random

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

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

In [3]:

```
# 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 [4]:

```
# 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 [5]:

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

Implement the regression function (normal equations)

Recall the normal equations used to find the $\theta$ 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 [6]:

```
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 [7]:

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

In [8]:

```
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(0.29154759474226505, 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 [9]:

```
# 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 $\theta$ as numpy vector `theta_pr` similarly to how we set up `theta_slr` above.

Hint here!

In [10]:

```
# 1. Add constant column and x^2 column
# 2. Find optimal parameters

# YOUR CODE HERE
X_aug = np.insert(X_aug, 2, X[:,0]**2, axis=1 )

theta_pr = regression(X_aug, y)
#raise NotImplementedError()
```

In [11]:

```
# 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 $\theta$ it is passed. The function should take as input a vector of scalar $x$ values along with a set of parameters $\theta$. 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 [12]:

```
def get_predictions(x, theta):
    # Change the shape of x to support the function
    x = np.array([x]).T
    x = np.insert(x, 0, 1, axis=1)
    while(x.shape[1] < theta.shape[0]):
        x = np.insert(x, x.shape[1], x[:,1] * x[:,-1], axis=1)
    y_hat = h(x, theta)
    # YOUR CODE HERE
    #raise NotImplementedError()

    return y_hat
```

In [13]:

```
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 None, "predict from theta_slr is incorrect"
assert np.round(get_predictions(np.array([1, 1, 0.1, 2]), theta_pr).T, 5) is not None, "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!
```

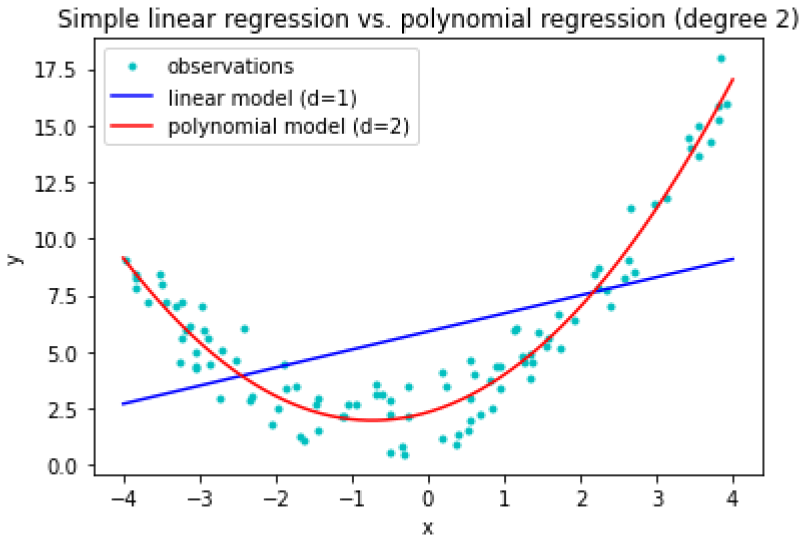
Expected output:\ y_series_slr: [[2.72462183 2.73101513 2.73740842]]\ y_series_pr: [[9.0812643 9.04632656 9.01147497]]

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 [14]:

```
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 [15]:

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

In [16]:

```
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 incorrect"
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.

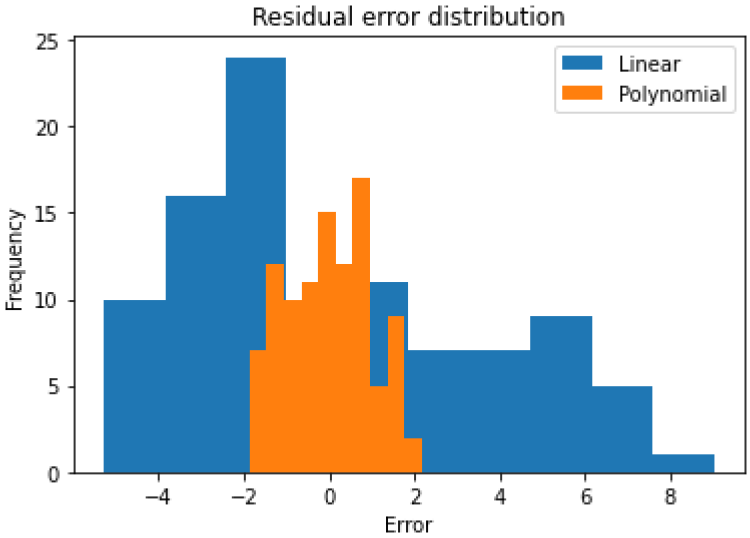
In [17]:

```
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 [18]:

```
# 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]])), "pr
edict from theta_slr is incorrect"
assert np.array_equal(np.round(get_predictions(np.array([0, 7, 1.5, -0.3]), theta_pr).T),
                      np.round([[2.34050076, 42.14663283, 5.3284002, 2.10566904]])), "pred
ict from theta_pr is incorrect"
print("success!")
# End Test function
```

```
error_slr sample: [-4.88494741 -0.58280848 -2.8007543  -5.27887921 -2.2790654
1]
error_pr sample: [-1.49521216  0.67105966  0.15715854 -1.86746535  1.1486978
5]
```



```
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 $\theta$, 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 [19]:

```
# 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)
```

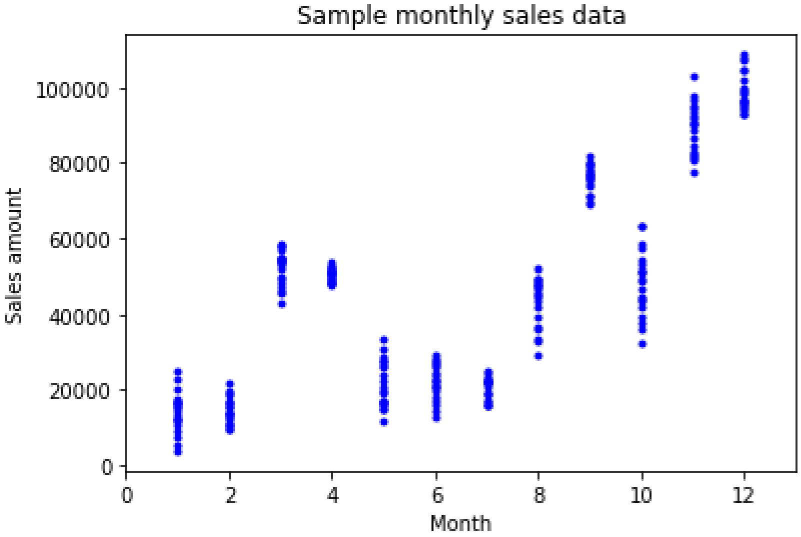
```
Headers: ['year' 'month' 'sale amount']
Extracted 240 monthly sales records
(240, 1)
(240,)
```

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 [20]:

```
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) </link> to shuffle the indices of the dataset.

In [21]:

```
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)
    m = X.shape[0]
    m_train = int(m * percent_train)
    train_idx = idx[:m_train]
    test_idx = idx[m_train:]
    X_train = X[train_idx, : ]
    X_test = X[test_idx, :]

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

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

In [22]:

```
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_train must be shuffled!"
assert not np.array_equal(np.round(X_data[144:, :], 3), np.round(X_test,3)), "X_test must be shuffled!"
assert not np.array_equal(np.round(y_data[0:144], 3), np.round(y_train,3)), "y_train must be shuffled!"
assert not np.array_equal(np.round(y_data[144:], 3), np.round(y_test,3)), "y_test must 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 [23]:

```
def x_polynomial(x, d):
    # YOUR CODE HERE
    X = np.ones((x.shape[0], 1))
    for i in range(d):
        X = np.concatenate((X, x**(i+1)),axis=1)
    return X
```

In [24]:

```
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 incorre
ct"
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)), "Polyno
mial 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 $\theta$.

Exercise 1.8 (2 points)

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

In [25]:

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

In [26]:

```
# 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 test set is wrong"
print("success!")
# End Test function
```

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

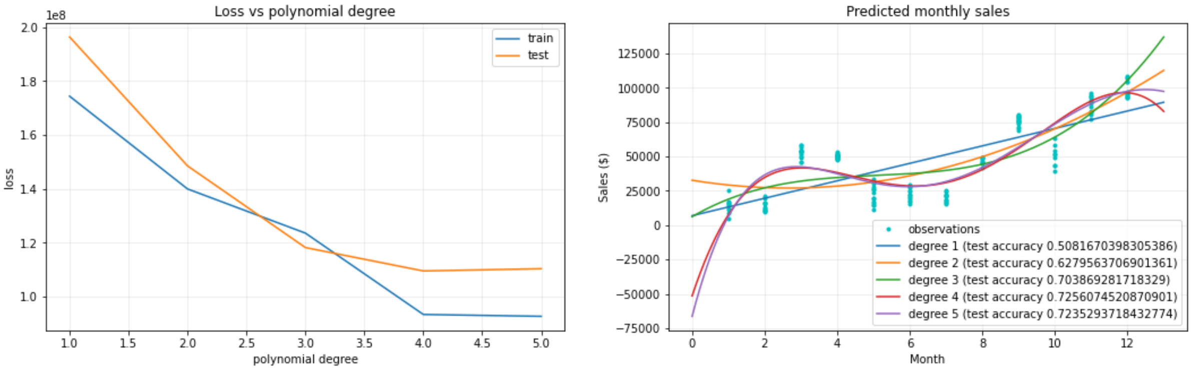
Expected output:\ J_train: 174395635.44334993\ J_test: 196382485.91395777

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 [27]:

```
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 [28]:

```
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 7.24420000
e+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 [29]:

```
# YOUR CODE HERE
data = np.genfromtxt('MonthlySales_data.csv', delimiter =',', dtype=str)

headers_csv = data[0,:]
data_csv = np.array(data[1:,:], dtype=float)
#raise NotImplementedError()
```

In [30]:

```
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 incorrect"
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 [31]:

```
# Extract y column from raw data
# Extract x column (year and month) from raw data
# YOUR CODE HERE
y_index = np.where(headers == 'sale amount')[0][0]
y_csv = data_csv[:, y_index]

X_csv = data_csv[:, 0:y_index]
#raise NotImplementedError()
```

In [32]:

```
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 wrong 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

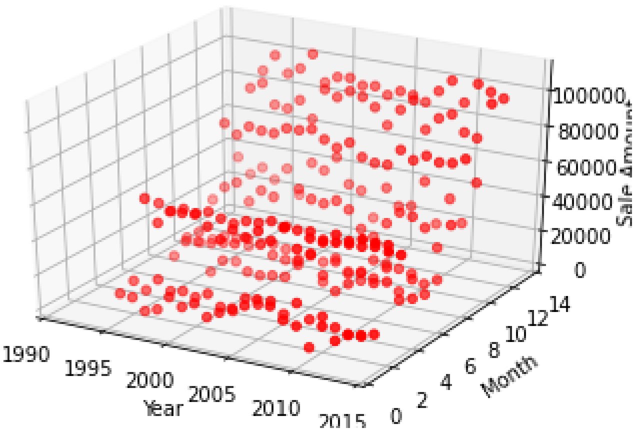
Exercise 2.3 (2 points)

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

Hint here!

In [33]:

```
# 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 = X_csv[:, 0]
y_month = X_csv[:, 1]
z_sale = y_csv

# 3. plot by using scatter
ax.scatter(x_year, y_month, z_sale, color='r')
# 4. set x, y, z label
# YOUR CODE HERE
ax.set_xlabel('Year')
ax.set_ylabel('Month')
ax.set_zlabel('Sale Amount', rotation=90)
ax.set_xlim(1990, 2015)
ax.set_ylim(0, 14)
#raise NotImplementedError()

plt.show()
```



In [34]:

```
# Test function: Do not remove
assert ax.get_xbound()[1] >= 2014 and ax.get_xbound()[0] <= 1995, "Year is filled wrong 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 wrong 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 [35]:

```
idx, X_train, y_train, X_test, y_test = partition(X_csv, y_csv, percent_train = 0.6)
# YOUR CODE HERE
#raise NotImplementedError()
```

In [36]:

```
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_train 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 must be shuffled!"
assert np.array_equal(idx[5:9], [26, 75, 51, 162])
print("success!")
# End Test function
```

```
(144, 2)
(144,)
(96, 2)
(96,)
[ 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 [37]:

```
Xi_train, Xi_test = x_polynomial(X_train, 1), x_polynomial(X_test, 1)
theta = regression(Xi_train, y_train)
y_pred_test = h(Xi_test, theta)
J_train, J_test = cost(theta, Xi_train, y_train), cost(theta, Xi_test, y_test)

# YOUR CODE HERE
#raise NotImplementedError()
```

In [38]:

```
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 incorrect"
assert np.round(J_test, 0) == np.round(204275431.7643744, 0), "Test cost is incorrect"
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: 172968387.44854635
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 [39]:

```
# 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
#     - do y_series in between min(month) - 1 to max(month) + 1
#     - 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_series, y_series = np.linspace(min(X_csv[:,0])-1, max(X_csv[:,0])+1, num_linspace),np.li
nspace(min(X_csv[:,1])-1, max(X_csv[:,1])+1, num_linspace)
x_mesh, y_mesh = np.meshgrid(x_series, y_series)
xy_mesh = np.dstack((x_mesh, y_mesh))
# 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
    return y
z_series = mesh_predictions(xy_mesh, theta).reshape(num_linspace, -1)
# YOUR CODE HERE
#raise NotImplementedError()
```

In [40]:

```
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)

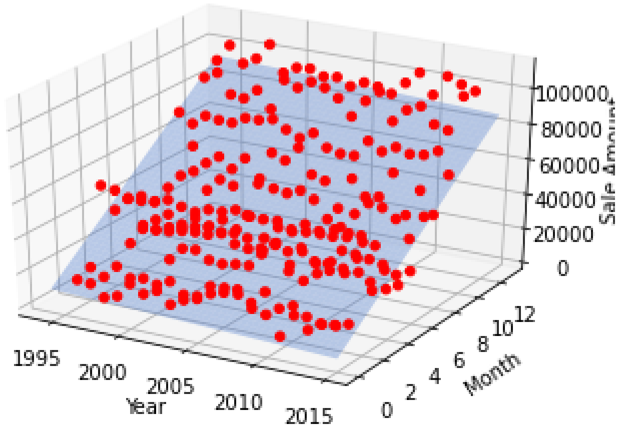
Exercise 2.6 (2 points)

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

Hint here!

In [41]:

```
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]
y_month = X_csv[:, 1]
z_sale = y_csv

# 3. plot by using scatter
ax.scatter(x_year, y_month, z_sale, color='r', alpha=1)
# 4. set x, y, z label
ax.set_xlabel('Year')
ax.set_ylabel('Month')
ax.set_zlabel('Sale Amount', rotation=90)
#    Hint: In these 3, 4 steps, you can copy Exercise 2.3
# 5. Plot surface from x_mesh, y_mesh, and z_series
ax.plot_surface(x_mesh, y_mesh, z_series, color='cornflowerblue', alpha=.4)
# YOUR CODE HERE
#raise NotImplementedError()

plt.show()
```



In [42]:

```
# Test function: Do not remove
assert ax.get_xbound()[1] >= 2014 and ax.get_xbound()[0] <= 1995, "Year is filled wrong 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 wrong 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.

In [43]:

```
# Your code here
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[43]:

```
((144, 2), (144,), (96, 2), (96,))
```

In [44]:

```
x = x_polynomial(X_train, 2)
print(x[:3])
```

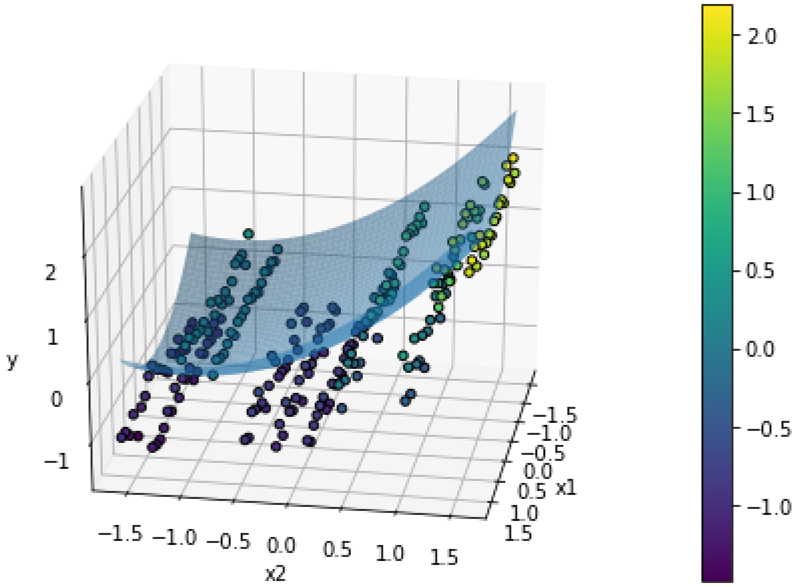
```
[[1.000000e+00 2.003000e+03 1.100000e+01 4.012009e+06 1.210000e+02]
 [1.000000e+00 2.004000e+03 3.000000e+00 4.016016e+06 9.000000e+00]
 [1.000000e+00 2.002000e+03 6.000000e+00 4.008004e+06 3.600000e+01]]
```

degree 2

In [45]:

```
data_norm =(data_csv - np.mean(data_csv, axis=0))/ np.std(data_csv, axis=0)
data_norm.shape
y_label = 'sale amount'
y_index = np.where(headers==y_label)[0][0]
y_csv = data_norm[:, y_index]

X_csv = data_norm[:, 0:y_index]

m = X.shape[0]
n = X.shape[1]

idx, X_train, y_train, X_test, y_test = partition(X_csv, y_csv, percent_train = 0.6)

max_degree= 2

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

for i in range(1, max_degree+1):

    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)

print(theta.shape)
print(Xi_train[:3])
fig = plt.figure()
ax = Axes3D(fig)

x_year = X_csv[:,0]
x_month = X_csv[:, 1]
y_sale = y_csv

p = ax.scatter(x_year, x_month, y_sale, edgecolors='black', c=data_norm[:,2], alpha=1)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

X1 = np.linspace(min(x_year), max(x_year), len(y))
X2 = np.linspace(min(x_month), max(x_month), len(y))
xx1, xx2 = np.meshgrid(X1, X2)

yy = theta[0] + theta[1]*xx1 + theta[2]*xx2 + theta[3]*xx1*xx2 + theta[4]*(xx2**2 + xx1**2
)
ax.plot_surface(xx1,xx2,yy, alpha=0.5)

ax.view_init(elev=25, azim=10)
plt.colorbar(p)
plt.show()
```

```
(5,)
[[ 1.         -0.26013299  1.30357228  0.06766917  1.6993007 ]
 [ 1.         -0.086711   -1.01388955  0.0075188   1.02797203]
 [ 1.         -0.43355498 -0.14484136  0.18796992  0.02097902]]
```

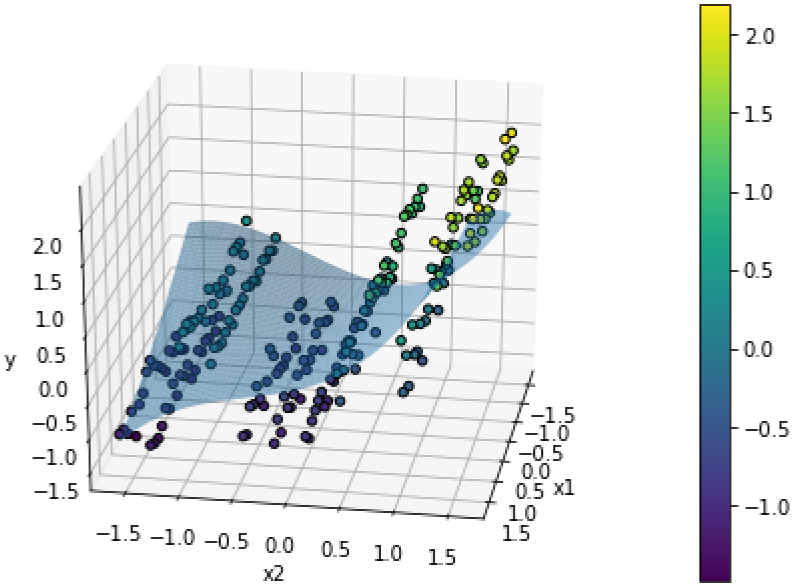


degree = 3

In [46]:

```
max_degree= 3

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

for i in range(1, max_degree+1):

    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)

print(theta.shape)
print(Xi_train[:3])
fig = plt.figure()
ax = Axes3D(fig)

x_year = X_csv[:,0]
x_month = X_csv[:, 1]
y_sale = y_csv

p = ax.scatter(x_year, x_month, y_sale, edgecolors='black', c=data_norm[:,2], alpha=1)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

X1 = np.linspace(min(x_year), max(x_year), len(y))
X2 = np.linspace(min(x_month), max(x_month), len(y))
xx1, xx2 = np.meshgrid(X1, X2)

yy=(theta[0]+theta[1]*(xx1+xx2).T+theta[2]*xx1*xx2+theta[3]*xx1**2+theta[4]*xx2**2+theta[5
]*xx2*xx1**2+theta[6]*xx2**3)
ax.plot_surface(xx1,xx2,yy, alpha=0.5)

ax.view_init(elev=25, azim=10)
plt.colorbar(p)
plt.show()
```

```
(7,)
[[ 1.00000000e+00 -2.60132991e-01  1.30357228e+00  6.76691729e-02
   1.69930070e+00 -1.76029843e-02  2.21516129e+00]
 [ 1.00000000e+00 -8.67109970e-02 -1.01388955e+00  7.51879699e-03
   1.02797203e+00 -6.51962383e-04 -1.04225010e+00]
 [ 1.00000000e+00 -4.33554985e-01 -1.44841365e-01  1.87969925e-01
   2.09790210e-02 -8.14952979e-02 -3.03863003e-03]]
```



From the 3D plot, we can see that we observe overfitting for the polynomial degree above 2.

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.

In [47]:

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

In [48]:

```
data = pd.read_csv('insurance.csv')
data.head(5)
```

Out[48]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

In [49]:

```
print(data.shape)
```

```
(1338, 7)
```

In [50]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1338 non-null   int64  
 1   sex       1338 non-null   object 
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64  
 4   smoker    1338 non-null   object 
 5   region    1338 non-null   object 
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

In [51]:

```
data.describe()
```

Out[51]:

	age	bmi	children	charges
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.663397	1.094918	13270.422265
std	14.049960	6.098187	1.205493	12110.011237
min	18.000000	15.960000	0.000000	1121.873900
25%	27.000000	26.296250	0.000000	4740.287150
50%	39.000000	30.400000	1.000000	9382.033000
75%	51.000000	34.693750	2.000000	16639.912515
max	64.000000	53.130000	5.000000	63770.428010

In [52]:

```
data.isnull().sum()
```

Out[52]:

```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```

Since there is no null value, we don't need to fill any missing data.

Preparing Data for the model

First, we will convert all values to type int or float so that the model can process them. (some columns have object as dtypes)

Categorical Features

- sex
- smoker
- region

In [53]:

```
print(data['sex'].unique())
print(data['smoker'].unique())
print(data['region'].unique())
```

```
['female' 'male']
['yes' 'no']
['southwest' 'southeast' 'northwest' 'northeast']
```

In [54]:

```
data['sex'].replace('male', 0.0, inplace = True)
data['sex'].replace('female', 1.0, inplace = True)
```

In [55]:

```
data['smoker'].replace('no', 0.0, inplace = True)
data['smoker'].replace('yes', 1.0, inplace = True)
```

In [56]:

```
data['region'].replace('northeast', 1.0, inplace = True)
data['region'].replace('southeast', 2.0, inplace = True)
data['region'].replace('southwest', 3.0, inplace = True)
data['region'].replace('northwest', 4.0, inplace = True)
```

In [57]:

```
data.head(5)
```

Out[57]:

	age	sex	bmi	children	smoker	region	charges
0	19	1.0	27.900	0	1.0	3.0	16884.92400
1	18	0.0	33.770	1	0.0	2.0	1725.55230
2	28	0.0	33.000	3	0.0	2.0	4449.46200
3	33	0.0	22.705	0	0.0	4.0	21984.47061
4	32	0.0	28.880	0	0.0	4.0	3866.85520

In [58]:

```
data.info()
```

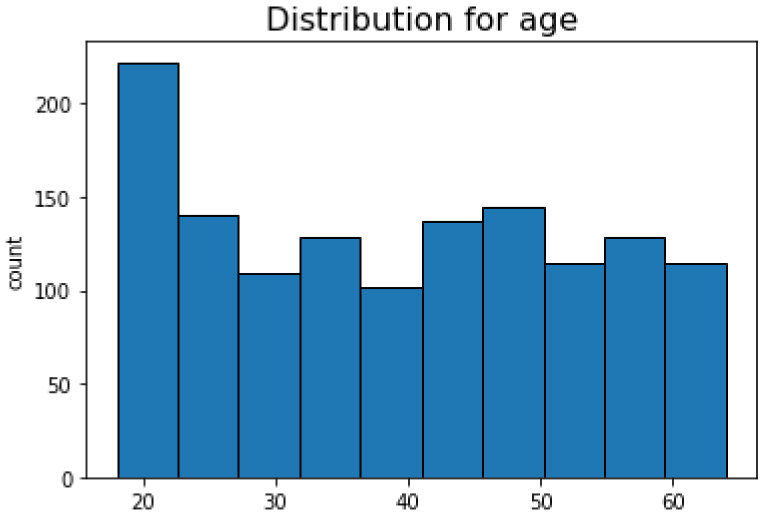
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1338 non-null   int64  
 1   sex       1338 non-null   float64
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64  
 4   smoker    1338 non-null   float64
 5   region    1338 non-null   float64
 6   charges   1338 non-null   float64
dtypes: float64(5), int64(2)
memory usage: 73.3 KB
```

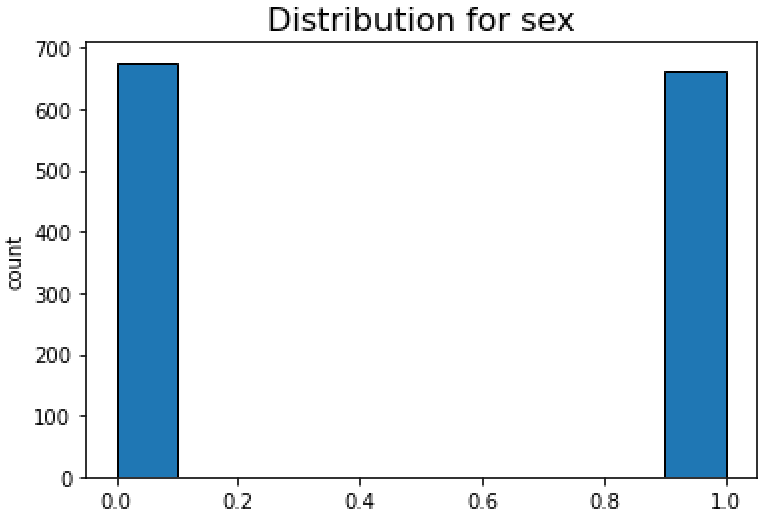
In [59]:

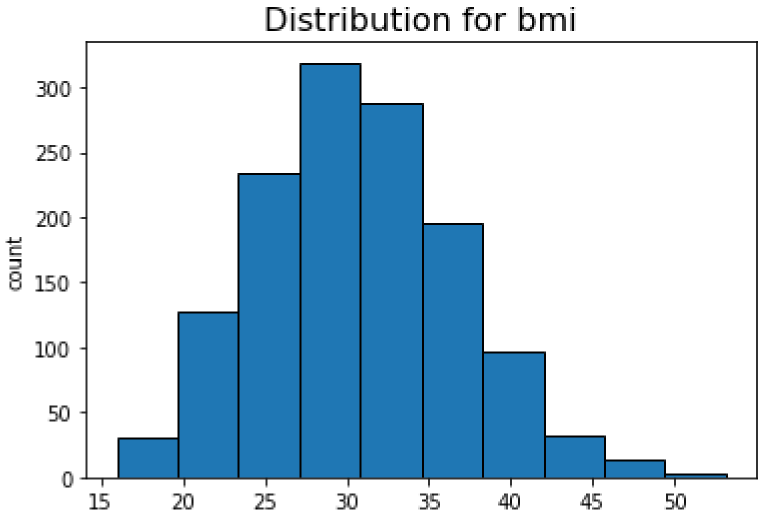
```
def hisplot(data, name):
    plt.hist(data[name], edgecolor='black')
    plt.title(f"Distribution for {name}", size=16)
    plt.ylabel('count')
    plt.show()
```

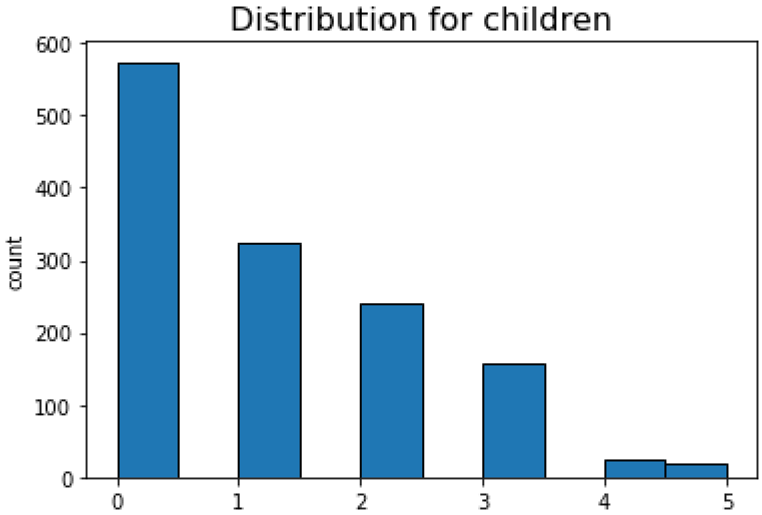
In [60]:

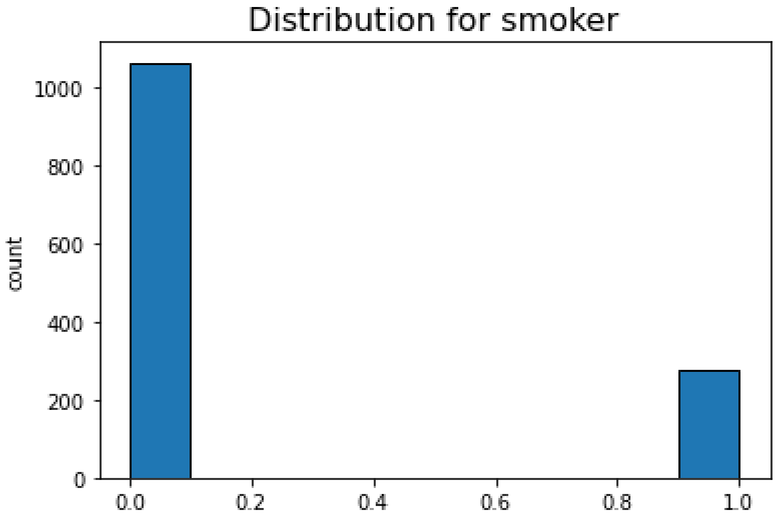
```
columns_name = list(data.columns)

for name in columns_name:
    hisplot(data, name)
```

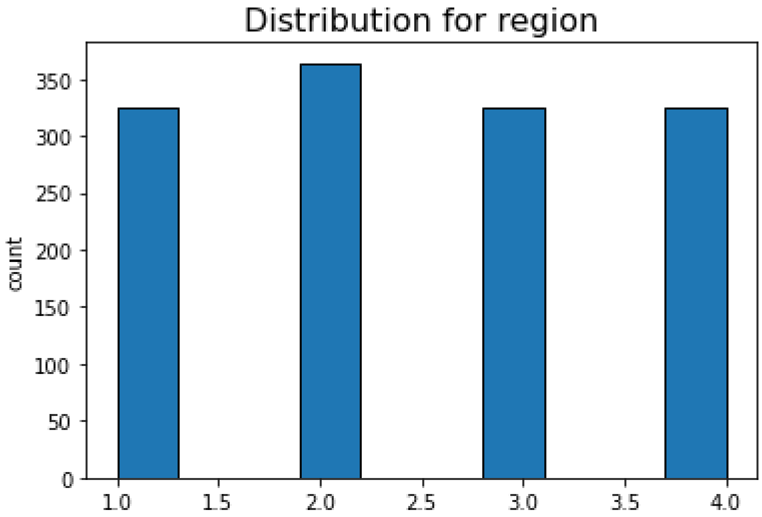

Distribution for age
count
0
50
100
150
200
20
30
40
50
60

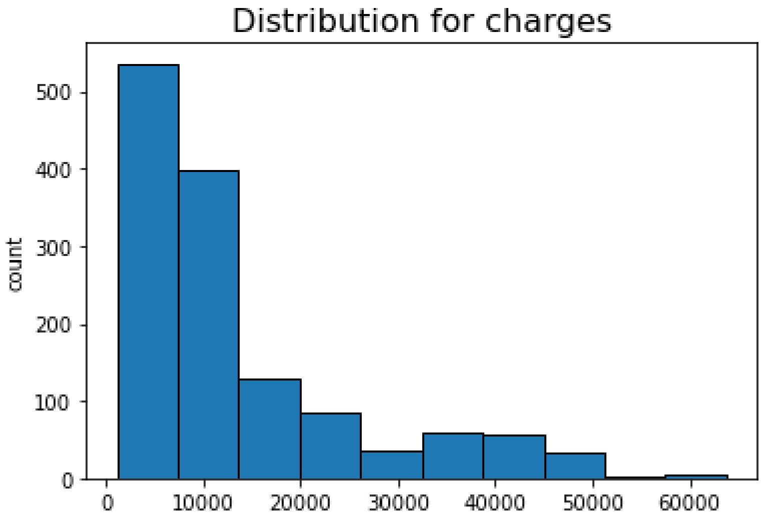

Distribution for sex
count
0
100
200
300
400
500
600
700
0.0
0.2
0.4
0.6
0.8
1.0


Distribution for bmi
count
0
50
100
150
200
250
300
15
20
25
30
35
40
45
50


Distribution for children
count
0
100
200
300
400
500
600
0
1
2
3
4
5


Distribution for smoker
count
0
200
400
600
800
1000
0.0
0.2
0.4
0.6
0.8
1.0


Distribution for region
count
0
50
100
150
200
250
300
350
1.0
1.5
2.0
2.5
3.0
3.5
4.0



Normalizing the data

In [61]:

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

In [62]:

```
data_norm.head(5)
```

Out[62]:

	age	sex	bmi	children	smoker	region	charges
0	-1.438764	1.010519	-0.453320	-0.908614	1.970587	0.464873	0.298584
1	-1.509965	-0.989591	0.509621	-0.078767	-0.507463	-0.440513	-0.953689
2	-0.797954	-0.989591	0.383307	1.580926	-0.507463	-0.440513	-0.728675
3	-0.441948	-0.989591	-1.305531	-0.908614	-0.507463	1.370259	0.719843
4	-0.513149	-0.989591	-0.292556	-0.908614	-0.507463	1.370259	-0.776802

In [63]:

```
y = data_norm['charges']

X = data_norm.drop(columns='charges', axis=1)
```

In [64]:

```
X = np.array(X)
y = np.array([y]).T

m = X.shape[0]
n = X.shape[1]

print(X.shape)
print(y.shape)
```

```
(1338, 6)
(1338, 1)
```

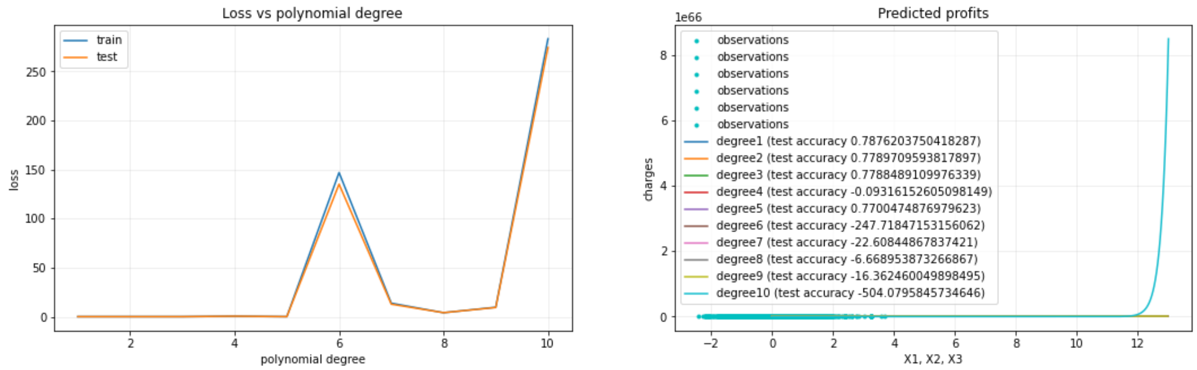
In [65]:

```
# Splitting training and test set
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(936, 6)
(402, 6)
(936, 1)
(402, 1)
```

In [66]:

```
max_degree = 10

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

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):

    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)

    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 profits')
plt2.set_xlabel('X1, X2, X3')
plt2.set_ylabel('charges')
plt2.grid(axis='both', alpha=.25)
plt2.legend()
plt.show()
```



The accuracy is the best in degree 1 and the accuracy decrease only 0.008 until degree 3. As we go further, we observe decreasing in accuracy. We can observe overfitting after degree 5.

In [ ]: