In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import scipy.io as sio
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from scipy.optimize import minimize
```

First load the data

In [2]:
```
# scipy.io, here loaded as 'sio' is a function to load data from a matlab file
data = sio.loadmat('sim.mat')
```

In [3]:
```
d = data['Data']
```

Split the data into test, train, and validation sets

In [4]:
```
train = np.concatenate((d[0:500], d[1500:2000], d[3000:3500], d[4500:]))
```

In [5]:
```
test = np.concatenate((d[500:1000], d[2000:2500], d[3500:4000]))
```

In [6]:
```
val = np.concatenate((d[1000:1500], d[2500:3000], d[4000:4500]))
```

In [7]:
```
x_train = np.delete(train, 34, axis=1)
y_train = train[:, 34]
```

In [8]:
```
x_test = np.delete(test, 34, axis=1)
y_test = test[:, 34]
```

In [9]:
```
x_val = np.delete(val, 34, axis=1)
y_val  = val[:, 34]
```

Use `scikit-learn`'s normalization

In [10]:
```
x_scaler = StandardScaler()
y_scaler = StandardScaler()
```

In [11]:
```
X_train = x_scaler.fit_transform(x_train)
Y_train = y_scaler.fit_transform(y_train.reshape(-1, 1))
```

Traditional (single regularization) Ridge Regression

In [12]:
```
simple_ridge = linear_model.Ridge(alpha=10.0)
```

In [13]:
```
simple_ridge.fit(X_train, Y_train)
```

Out[13]:
```
Ridge(alpha=10.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

Check the error

In [14]:
```
X_test = x_scaler.transform(x_test)
```

In [15]:
```
mean_squared_error(y_val, y_scaler.inverse_transform(simple_ridge.predict(X_te
st)))
```

Out[15]:
```
0.66194072929083281
```

Concept proven; now calculate MSE for a variety of α^2 values

To determine the range of values to check for alpha we find the singular values

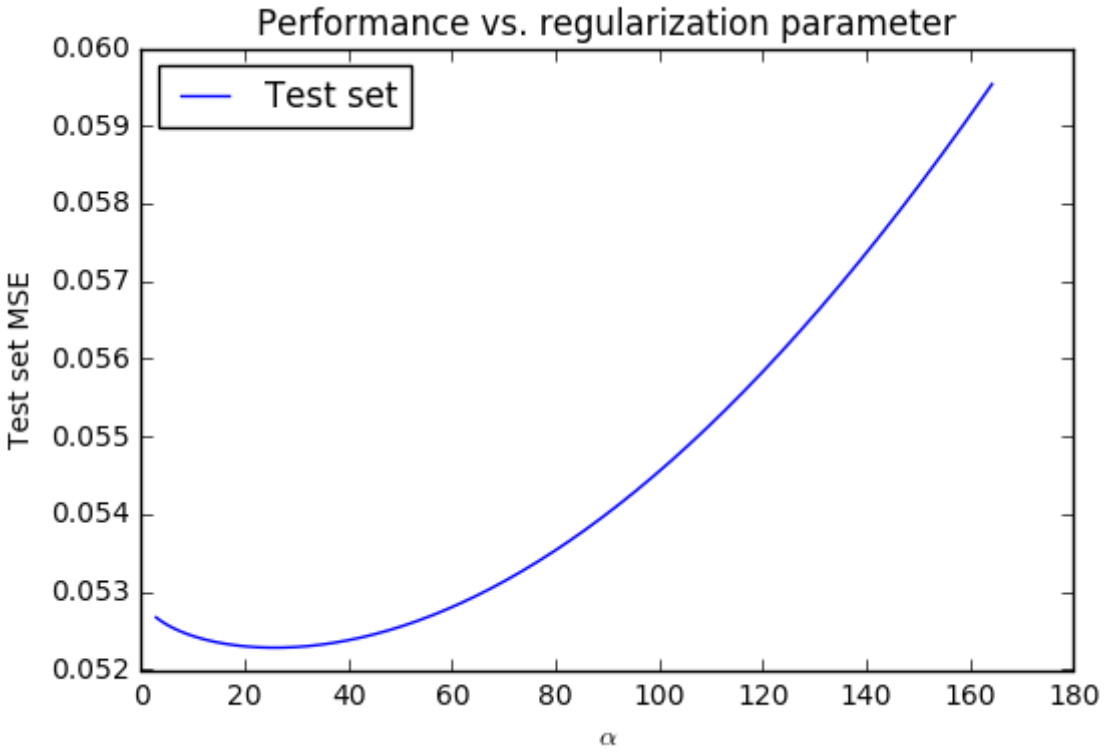
In [16]:
```
U, s, Vh = np.linalg.svd(X_train)
alpha_min = min(s)
alpha_max = max(s)

print([alpha_min, alpha_max])
```

```
[2.8860179913035178, 164.24424930494501]
```

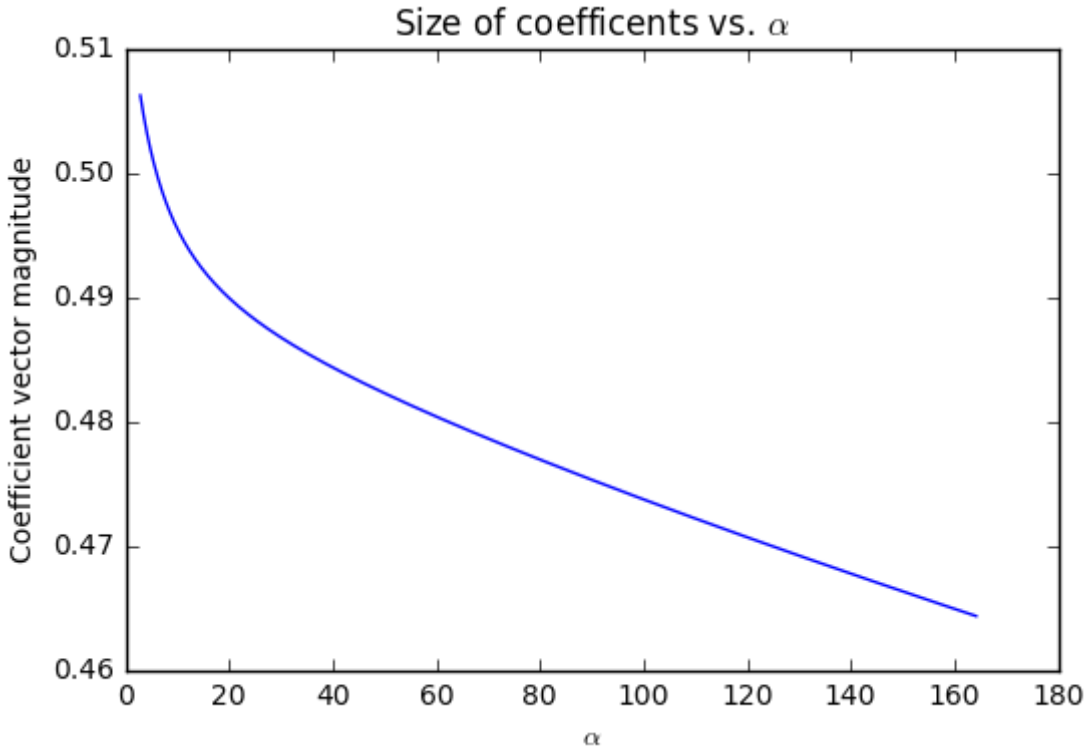
In [17]:
```
# ~Equivalent to Matlab 0:0.1:20
alphas = np.linspace(alpha_min, alpha_max, 1000)
```

In [18]:
```
mse = []
b_mag = []
for a in alphas:
    mdl = linear_model.Ridge(alpha=a)
    mdl.fit(X_train, Y_train)
    m = mean_squared_error(y_test, y_scaler.inverse_transform(mdl.predict(X_te
st)))
    mse.append(m)
    b = np.linalg.norm(mdl.coef_)
    b_mag.append(b)
```

In [19]:
```
plt.plot(alphas, np.array(mse), label='Test set')
plt.title(r'Performance vs. regularization parameter')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'Test set MSE')
plt.legend(loc='upper left')
plt.savefig('images/simple_alpha_full.png', dpi=300)
plt.show()
```



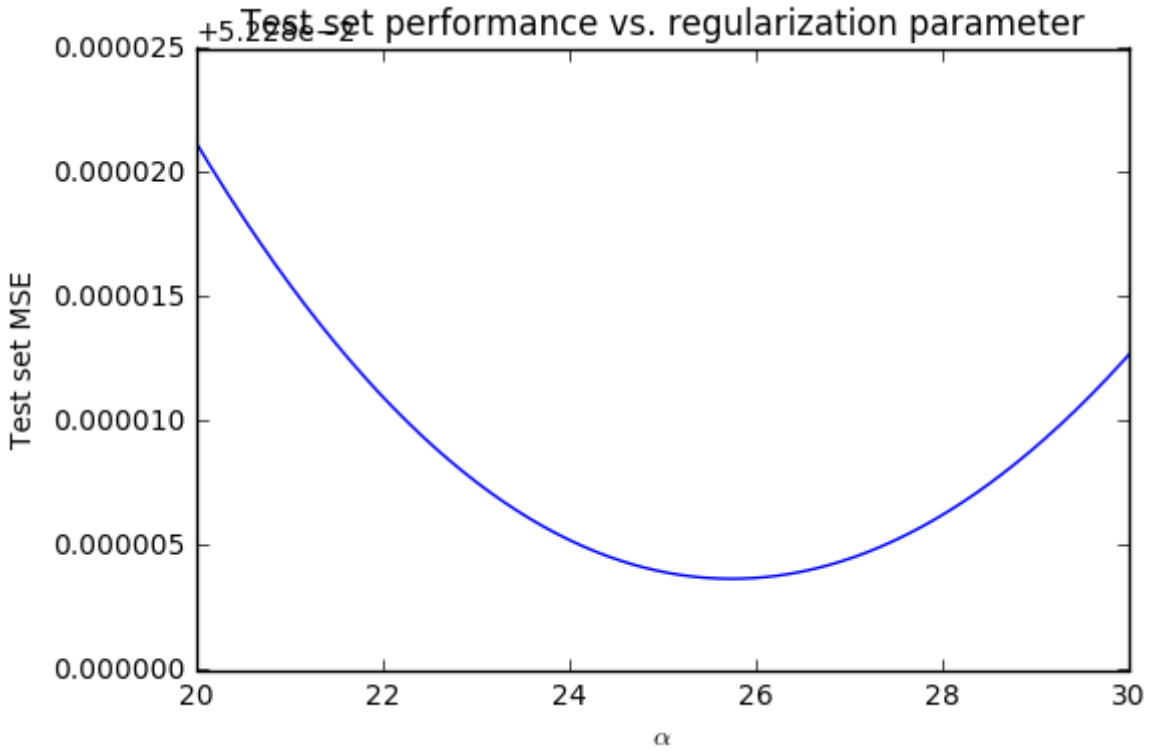
In [20]:
```
plt.plot(alphas, np.array(b_mag))
plt.title(r'Size of coefficents vs. $\alpha$')
plt.xlabel(r'$\alpha$')
plt.ylabel('Coefficient vector magnitude')
plt.savefig('images/simple_coeff_mag.png', dpi=300)
plt.show()
```



Here the best value for α appears to be between 20 and 30

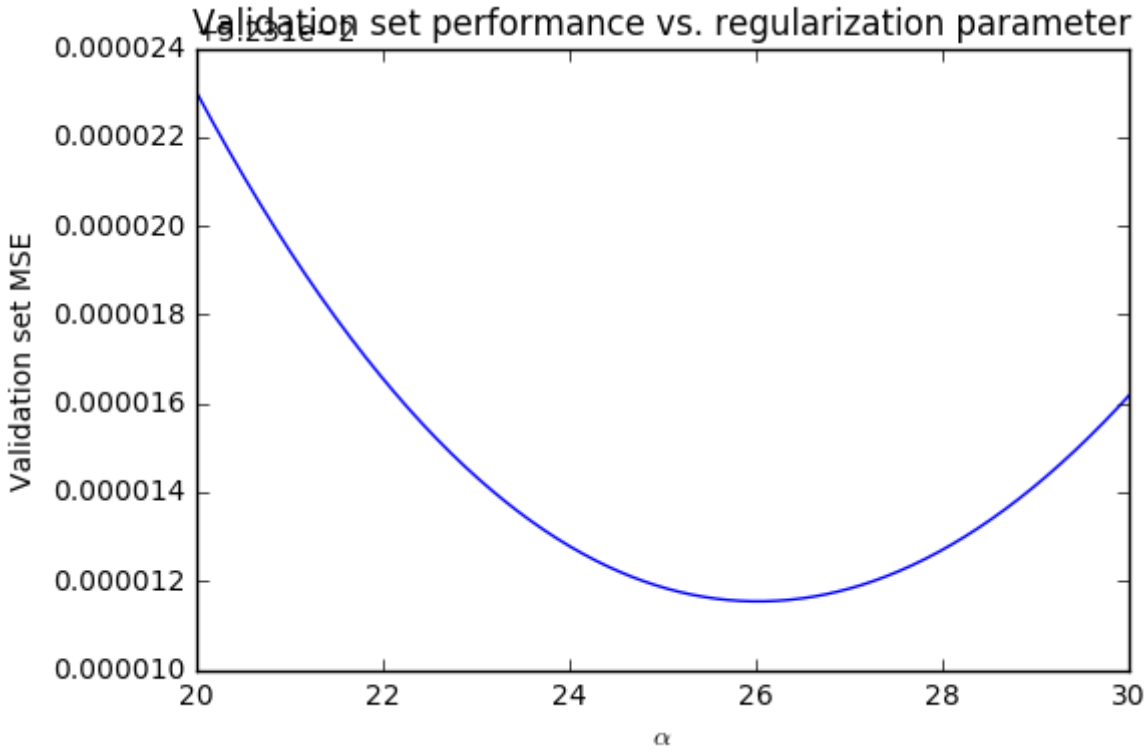
In [21]:
```
alphas2 = np.linspace(20, 30, 1000)
mse2 = []
for a in alphas2:
    mdl = linear_model.Ridge(alpha=a)
    mdl.fit(X_train, Y_train)
    m = mean_squared_error(y_test, y_scaler.inverse_transform(mdl.predict(x_sc
aler.transform(x_test))))
    mse2.append(m)
```

In [22]:
```
plt.plot(alphas2, np.array(mse2), label='Test set')
plt.title(r'Test set performance vs. regularization parameter')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'Test set MSE')
plt.savefig('images/simple_alpha.png', dpi=300)
plt.show()
```



In [23]:
```
alphas3 = np.linspace(20, 30, 1000)
mse3 = []
for a in alphas3:
    mdl = linear_model.Ridge(alpha=a)
    mdl.fit(X_train, Y_train)
    m = mean_squared_error(y_val, y_scaler.inverse_transform(mdl.predict(x_sca
ler.transform(x_val))))
    mse3.append(m)
```

In [24]:
```
plt.plot(alphas2, np.array(mse3), label='Test set')
plt.title(r'Validation set performance vs. regularization parameter')
plt.xlabel(r'$\alpha$')
plt.ylabel(r'Validation set MSE')
plt.savefig('images/simple_alpha_val.png', dpi=300)
plt.show()
```



We can also use the `RidgeCV` estimator to optimize α

This implements "Leave One Out Cross Validation"

In [25]:
```
simple_ridge2 = linear_model.RidgeCV(alphas=np.linspace(20, 30, 1000))
```

In [26]:
```
simple_ridge2.fit(X_train, Y_train)
```

Out[26]:
```
RidgeCV(alphas=array([ 20.     ,  20.01001, ...,  29.98999,  30.     ]),
    cv=None, fit_intercept=True, gcv_mode=None, normalize=False,
    scoring=None, store_cv_values=False)
```

In [27]:
```
simple_ridge2.alpha_
```

Out[27]:
```
20.860860860860861
```

Here we see that the data in the training set is slightly different from that in the test set. Leave one out cross validation optimizes alpha based on values in the training set, explaining the difference in the values from each method.

Now we implement the local ridge regression method

In [28]:
```
np.linalg.cond(x_train)
```

Out[28]: 102.94187511494458

In [29]:
```
np.linalg.cond(X_train)
```

Out[29]: 56.910334516231281

Set up our objective function

In [36]:
```
def b(x, y, alpha):
    x = np.matrix(x)
    U, v, Vh = np.linalg.svd(x)
    Vht = np.matrix.transpose(Vh)
    xt = np.matrix.transpose(x)
    #tmp = np.dot(xt, x) + np.dot(np.dot(Vht, alpha ** 2), Vh)
    tmp = (xt * x) + np.diag(alpha **2)
    tmp = np.linalg.inv(tmp)
    tmp = tmp * xt
    return np.dot(tmp, y)
```

In [37]:
```
obj = lambda alpha: mean_squared_error(y_test, y_scaler.inverse_transform(np.d
ot(X_test, b(X_train, Y_train, alpha))))
```

In [ ]:
```
minimize(obj, 40*np.ones(43), method='CG')
```

In [ ]:
```

```